Abstract—This paper presents HyperFlow: an approach to workflow programming which combines the advantages of a declarative workflow description and low-level scripting programming. The workflow execution model of HyperFlow is based on a formal model of computation – Process Networks. The execution environment is implemented on the basis of a widely adopted runtime platform node.js. Workflow programming benefits from such an approach in multiple ways, including leveraging a large programming ecosystem with many developers, reusable software packages and learning resources; elimination of shim nodes from the workflow graph; and increased reusability of workflow processing components. The HyperFlow workflow programming approach and its advanced capabilities are presented. The HyperFlow engine is also briefly described. Four example workflow applications from various domains, including flood threat assessment, are studied as a demonstration of the HyperFlow programming approach and a comparison with related solutions.

I. Introduction

Visual programming is a dominating approach in scientific workflow development because of its suitability for domain experts who have limited programming skills. It also naturally leads to a graph-based workflow model conducive to implicit expression of parallelism, well-defined data dependencies, and reproducibility, all important in scientific experiments. Graphical workflow programming environments consist of a visual programming language and a library of processing components which can be composed as workflow activities, much like a traditional programming language provides statements and libraries. On the other hand, visual programming often leads to overly complex workflow models, obscured by ‘shim’ nodes which represent the glue code. Also, the processing components which can be invoked from workflow activities tend to be tightly coupled with the runtime environment of a particular workflow system, which hinders their reusability.

A different group of approaches is addressed to workflow developers who are expert programmers. They prefer script-based programming languages because scripts are conducive to a more compact workflow representation and rapid development. Script-based workflow programming languages that have emerged, rather than being based on well-known general purpose scripting languages, are custom-designed in order to preserve the advantages of a well-defined graph model, or even mimic its semantics [1], [2].

Among key factors that contribute to a high programming productivity are the features of the programming language itself (such as concise and intuitive syntax), but also the maturity of the entire programming ecosystem, characterized by a large developer community, tens of thousands of reusable software packages, and many learning resources. From this perspective, both visual and script-based workflow programming languages are characterized by small ecosystems when compared to one of any mainstream programming language.

We present HyperFlow: an approach to workflow programming and enactment which combines the benefits of a well-defined graph-based workflow model with those associated with a low-level scripting programming language. The HyperFlow workflow programming model is based on two pillars: a simple declarative workflow description language which defines the workflow graph, and the implementations of workflow activities in JavaScript / node.js. Such a separation allows to preserve the desired properties associated with a graph-based model while gaining the advantages of a widely adopted scripting programming language and its ecosystem. The workflow structure is defined in terms of just three abstractions: processes, signals, and functions. However, despite the simplicity, the model (based on Process Networks as the underlying model of computation) is sufficiently expressive even for complex workflow patterns found in scientific workflows.

The paper is organized as follows. Section II overviews related work. Section III presents the HyperFlow workflow model and description language, while Section IV describes the HyperFlow workflow engine. In section V selected advanced workflow programming capabilities are discussed. Section VI presents four
case study workflows implemented in HyperFlow. Section VII concludes the paper.

II. Related work

A typical workflow system based on the visual programming paradigm provides a graphical programming environment comprising a visual language and a large base (several hundreds) of built-in processing components which can be composed as activities in the workflow graph. These components have different names in workflow systems but conceptually follow the same philosophy, e.g. activities in ASKALON [3], actors in Kepler [4], services in Taverna [5], or tools in Triana [6]. The available processing components provide the workflow programmer with various capabilities, such as control flow constructs (e.g. loops, switch construct), basic operations (e.g. string manipulation and array operations), as well as high-level functions analogous to libraries, e.g. HTTP and Web Service clients, file operations, wrappers for specific tools, such as R, or other domain-specific components. While such an approach is convenient for a domain expert who possesses only limited programming skills, for an expert programmer it poses some disadvantages which add complexity to the workflow development process.

Firstly, even very simple operations have to be implemented in a dedicated component technology of a given workflow system. For example, in ASKALON where workflows are composed of abstract activities, a simple data conversion requires a dedicated Data Conversion Activity which needs to be semantically described and registered in an activity registry.

Secondly, low-level operations typical to glue code become part of the workflow model as so called 'shim' nodes. Consequently, the workflow graph can easily be obliterated by a low-level glue code logic, tedious to program in a visual manner. The workflow model also becomes sensitive to changes of component interfaces: most likely they also entail changes to the associated shim nodes, and perhaps the redesign of the whole workflow graph. In some cases there are ways of mitigating this, e.g. the Taverna’s ‘beanshell’ component can wrap a snippet of Java code. However, in practice Taverna is strongly oriented towards Web Services as core domain-specific components, while beanshell is specifically meant for auxiliary purposes, i.e. mainly for implementing shim nodes. The general principle is thus the same and the amount of shim nodes remains considerable.

Thirdly, the library of components a workflow programmer can use is limited to those currently provided in the workflow system by a relatively small community of developers. Also, the components are not reusable: those developed in one workflow system cannot be easily used in another one because their invocation is tightly coupled to the particular runtime environment (or even an entire ecosystem) of a given workflow system.

Notable workflow systems which follow a different philosophy are Pegasus [7] and WS-PGRADE [8], [9]. The workflow model in both these systems is based on a Directed Acyclic Graphs (DAG). Processing components are, in practice, constrained to external executable programs, while edges of the graph represent file transfers. WS-PGRADE also supports parameter studies as a special workflow class. Such a model is very effective in many applications, yet, because of its simplicity, limited in terms of programming capabilities.

The SHIWA workflow interoperability project [10] provides technology to convert between different workflow representations through a common Interoperable Workflow Intermediate Representation (IWIR) [11]. Significantly, SHIWA supports only this simplified workflow model where processing components are executable programs, i.e. can be executed independently of the runtime environment of a workflow system. Providing a deeper interoperability on the level of fine-grained processing components of a given workflow system would be much more difficult.

Finally, there are script-based approaches to programming scientific applications. SwiftScript [1] and gscript [2] both propose custom-designed languages based on the dataflow concurrency constructs, such as single-assignment future variables. Such a choice provides data-driven concurrency semantics, implicit parallelism, and well-defined data dependencies – features and benefits characteristic to a graph model (gscript is actually a script-based counterpart of the GWENDIA workflow language [12]). Still, from the perspective of a broader programming ecosystem, a custom-designed scripting language suffers from the same disadvantages as the visual workflow languages. The programmer is still constrained to those processing components which are built into the language and its runtime environment. For example, SwiftScript supports only execution of external programs, either locally or on remote sites, using a limited set of supported execution services. Gscript has the notion of invokers which can wrap various processing components, such as a local Java code, web service invocation, or a grid job submission. In any case, adding new capabilities, if possible, requires an extra development effort.

One can easily see that many of the above problems are mitigated or eliminated in an ecosystem of any mainstream programming language: a large number of libraries is readily available; the glue code can easily be added using rich programming constructs and libraries typical for a general-purpose programming language; component reuse only requires the runtime environment of a given language, not that of the workflow system; finally, the developer community is much larger. Having recognized these benefits, the authors of [13] propose an approach to programming
scientific applications based entirely on Ruby scripts. However, the lack of a well-defined workflow model and properties associated with it generates problems in other areas. For example, parallel programming is much more difficult, while provenance logging basically requires the tracking of variable dependencies (ibid.).

We propose an approach which combines the benefits of a well-defined declarative workflow description with those of low-level programming in a mainstream language. We made the following design choices: Kahn Process Networks [14] are the basis of the underlying model of computation (albeit some of the original constraints of PNs are relaxed). JavaScript/JSON and node.js [15] are the foundations of the workflow programming and execution environment, mainly because of the popularity of JavaScript even with non-professional programmers, and because of a huge and dynamically growing community of node.js. Notably, node.js is supported by all big cloud providers in terms of official client libraries (Google, Amazon, Azure) and in many cases also as one of available runtime environments (e.g. on Azure and Heroku).

III. Workflow model and description

A HyperFlow workflow is essentially a set of processes performing well-defined functions and exchanging signals. A process transforms input sequences of signals into output sequences using the associated function. Workflow programming involves writing a workflow description in JSON and implementation of its activities in JavaScript.

As a simple example, let us consider a workflow which consumes a sequence of numbers and produces a sum of squares of every three of them, in the order of arrival. Fig. 1 graphically illustrates this workflow, presents its HyperFlow model expressed in a JSON syntax, and its functions implemented in JavaScript. The workflow comprises two processes and three signals which are described in two arrays: processes and signals, respectively.

For each process of a workflow, we simply define its input signals, output signals, the associated function, and its type (which is a key feature of the model, described separately in Section V-A). The first process (Sqr) consumes numbers and computes their squares in parallel, owing to the parlevel:0 property which means that an unlimited number of process firings can run concurrently (see section V-D for more information). However, the ordering property is also specified, so the results are not emitted out of order. The second process (Sum) consumes every three squares (thanks to the square:3 signal quantity modifier, described in section V-B) and produces their sum.

A running workflow instance is essentially a collection of autonomous processes, each of which runs according to the following cycle: (i) receiving input signals, (ii) executing processes and their functions, (iii) ordering results, (iv) emitting output signals, and finally (v) repeating the cycle.

Fig. 1: Sum of squares workflow: the workflow comprises two processes and three signals, (a) Visual illustration of the sum of squares workflow, (b) HyperFlow description of the sum of squares workflow, (c) Functions used by the sum of squares workflow.
Fig. 2: HyperFlow data structures. Signal is a JSON object which contains metadata and an array of signal instances. Function inputs and outputs are passed as arrays of signal objects.

**Fig. 2 (left):** Structure of function ins / outs

**Fig. 2 (right):** Structure of a signal object

- **signals**: if the signals cannot be consumed immediately, they are buffered on the input queues of the process;
- (ii) **testing if a firing pattern is fulfilled**, i.e. all signals required for a firing have arrived;
- (iii) **invoking the function**: the signals required for firing are removed from the queues and passed as function parameters;
- (iv) **awaiting for the function’s callback**;
- (v) **emitting output signals**: the instances of the output signals are passed via the callback.

**Function** assigned to a process implements its activity and at the same time provides a layer of abstraction over the actual way the underlying processing component is implemented or invoked. A process upon firing simply passes input signals consumed in this firing to the function and retrieves output signals via a callback. This simple approach fits virtually any execution model: synchronous web service call, asynchronous job submission to an external execution service, coordinated invocation of multiple services, or waiting for a human interaction. Being implemented in JavaScript, functions are a convenient place to add any extra glue code without extending the workflow graph with additional shim nodes. Furthermore, functions do not rely on any HyperFlow-specific context, so their invocation requires only the node.js runtime environment, while being independent of the HyperFlow engine, which fosters their reusability.

**Signals** in HyperFlow are an abstraction of data and control flow between workflow processes. Note that the model does not utilize the concept of ports, typical in many workflow models, where ports denote data produced or consumed by workflow components. Signals, on the other hand, are an abstraction of information exchanged between components. This simplifies the workflow description language, because there is no need for naming ports and specifying their connections. The workflow design focuses on the information flow essential to the processing pipeline, not on composing individual services and reconciling their incompatible APIs – these aspects are hidden behind the function abstraction.

A signal is represented as a simple JSON structure presented in Fig. 2, right. A signal is composed of metadata and optional data, i.e. actual signal instances. The metadata includes the signal name, and any number of additional properties. Signal instances are arbitrary JSON objects provided in the `data` array.

The input signals are passed to the function as the `ins` parameter which is an array of signal objects (Fig. 2, left). The second function parameter `outs` has exactly the same structure, except it only contains metadata of the output signals. In order to produce the actual outputs, the function should fill in the `outs` array with signal instances, and invoke the callback. If signals returned by the function contain no `data` arrays, no signals will be emitted which is the simplest way to implement filtering.

HyperFlow intentionally does not impose any data type system, but provides means to adopt one on the basis of signal metadata which, optionally, may contain a `schema` object compliant with the JSON schema.\(^1\) If such a schema is provided, signal instances must conform to it and will be validated by the HyperFlow engine.

**IV. HyperFlow engine**

The HyperFlow workflow enactment engine is a lightweight node.js application (the core code base, excluding libraries and workflow-specific functions, is only about 200 kB). Besides the node runtime, HyperFlow also requires Redis [16] which serves as a persistent store for the state of workflow instances.

\(^1\)http://json-schema.org
The simplest way to use the HyperFlow engine is through its command line tool `hflow`. For example, a workflow can be executed as follows: `hflow run <wf_dir>`. The directory `wf_dir` should contain at least two files: `workflow.json` and `functions.js` which implement the workflow description and workflow functions, respectively.

Alternatively, the HyperFlow engine can be started in a server mode: `hflow start-server`. In such a case workflows can be managed via a HTTP REST API. Multiple workflow instances can be started by posting their description to the application factory URI of a HyperFlow server. Every workflow instance, as well as its processes and signals, also have their unique URIs.

Signals can be sent to a workflow instance either using the `hflow` command, or via the REST API. A third possibility is to provide signal instances directly in the workflow description. Such signals are called initial signals and will be emitted by the HyperFlow engine as soon as an instance of the workflow is created. For example, in the sum of squares workflow there are six initial instances of the `number` signal (Fig. 1). As a result, when executed the workflow will immediately produce two output signals: 14 and 77.

HyperFlow provides the possibility to link multiple workflow instances, even running under control of different engine instances, so that they can exchange signals. To this end, it is sufficient to add an URI of a remote workflow instance to a list of `remoteSinks` of a specific signal, as in the code snippet in Fig. 3. Remote sinks can also be updated dynamically, via the REST API. Owing to this mechanism, workflows can be executed in a completely distributed and decentralized manner, without the control of a single enactment engine. See the `DistributedPingPong` example in the HyperFlow distribution\(^7\) for a more complete illustration. Workflow management and distributed execution enabled by a REST API was inspired by our earlier work on modeling workflows as hypermedia applications [17].

\(^7\)http://github.com/dice-cyfronet/hyperflow/tree/master/examples/DistributedPingPong

Fig. 3: Enabling distributed and decentralized workflow execution. URI of a workflow instance is registered in remote sinks of a signal. As a result, whenever the Pong signal is produced, it will be sent via HTTP to a remote workflow.

The HyperFlow engine also automatically logs provenance data in the form of events compliant with the Read – Write – State-Reset (RWS) model [18]. A log of such events can be translated to a structured data model and imported to a database in order to enable complex provenance queries.

The HyperFlow engine is available as open source at http://github.com/dice-cyfronet/hyperflow.

V. Advanced programming capabilities

A. Process types

A key feature of the model is the process type which characterizes the general behavior of the process. Providing new process types is also the easiest way to extend the model with new capabilities while preserving backward compatibility. Over the course of HyperFlow development, the following process types have been devised: dataflow, choice, foreach, and join. They are characterized below.

**Dataflow.** A dataflow process waits for all input signals, invokes the function, and emits the output signals returned by the function via a callback.

**Foreach.** A foreach process waits for any of its input signals, invokes its function passing this signal, and emits the corresponding output signal.

**Choice.** A choice process behaves similarly to dataflow but in each firing it may emit only some or none of its output signals. To this end, the function of the choice process must explicitly set a flag to denote which of the outputs should be emitted. This behavior is very useful for such patterns as conditional execution, data filtering or data routing.

**Join.** A join process joins/merges parallel branches of execution represented by its input signals. The behavior of a join process is governed by two parameters:

- \(N_b\) (active branches count): how many input branches out of total \(N\) (\(N_b \leq N\)) are active?
- \(N_j\) (join count): how many branches to wait for before firing? (\(N_j \leq N_b\))

For example, a join process with \(N = 4\), \(N_b = 3\), \(N_j = 1\) will work according to the following steps:

1) wait for one (\(N_j\)) signal on any input, and fire (invoke the function).
2) wait for the signals on the two remaining active branches (\(N_j\) total), and discard these signals.
3) reset the process, i.e. go back to step 1.

The following five patterns (described on http://www.workflowpatterns.org) are enabled by the Join process type. Four of them are illustrated in Fig. 4.

1) **Structured Discriminator.** The Join process with parameters \(N_j = 1\), \(N_b = \#\) of inputs enables this pattern.
2) **Blocking Discriminator.** Same as structured discriminator with additional next signal emitted by the join process after reset.

3) **Structured Partial Join.** A Join process fulfills this pattern with its parameters $N_j$ – number of branches to join, $N_b$ – number of active branches.

4) **Blocking Partial Join.** Same as Structured Partial Join with additional next signal emitted by the join process after reset.

5) **Local Synchronizing Merge.** This pattern is made possible by the choice and join processes connected with the merge control signal (see section V-C). This signal sets the $N_b$ and $N_j$ parameters of the join process to the number of branches enabled by the choice process in a given firing.

### B. Signal quantity

A number of important workflow patterns require information about the quantity of data elements consumed or produced by workflow activities. In the simplest case, it may be specified how many input signals must be consumed before a process can fire, as in the Sum process of the sum-of-squares workflow: `{"ins": "square:3"}`. In a more complex case, the quantity varies from firing to firing and may only be known at runtime. Let us consider a typical workflow pattern (Fig. 5), where process $P_1$ produces a collection of elements (sig1), process $P_2$ processes them one by one, while process $P_3$ has to collect all results (sig2) before the workflow can proceed. In HyperFlow such a behavior is achieved simply by adding `sig1:sigcount` to $P_1$’s outputs and `sig2:sigcount` to $P_3$’s inputs (where `sigcount` is actually a name of a ‘count’ control signal). As shown in section VI-B, this mechanism provides a simple and powerful syntax for processing data collections.

### C. Control signals

Apart from regular data signals, the HyperFlow model also features so called control signals. Control signals are not passed to the function but they affect the general behavior of the process which consumes them. A signal with property `control:type` is a control signal where `type` denotes its type. The following control signal types are currently defined in the HyperFlow workflow model:

**Next.** If added to inputs of a process, the next signal is required for firing along with other data signals; as an output of a process, next signal will be emitted after each firing of the process. This semantics enables synchronous loop-like execution of any subgraph of a workflow.

**Done.** If added to inputs of a process, the done signal will cause the process to terminate right after the current firing has finished; as an output, it will be emitted as a last signal prior to the termination of a process.

**Count.** A count signal is always associated with another data signal. As an output, the count signal will be emitted each time the associated data signal is emitted and will contain the number of produced signal instances of the associated data signal. On input, the count signal denotes the number of instances of the associated data signal required for the next firing of the process.


**Merge.** The merge signal may only be emitted by a choice process and consumed by a join process. The merge signal denotes the number of branches activated by a choice process in a given firing, and the number of branches required for firing of a join process.

**D. Parallel processing**

The HyperFlow model supports all basic types of parallelism: task parallelism, data parallelism and pipelining.

**Task parallelism.** Task parallelism simply means that distinct tasks run in parallel. In HyperFlow this type of parallelism is implicit: a process is fired as soon as signals required for the firing are available.

**Data parallelism.** This form of parallelism occurs when the same task is executed in parallel for multiple data elements. HyperFlow provides two idioms for data parallelism. The first one are two processes connected by a signal. The first process produces multiple instances of the signal (a collection) in a single firing, while the second one consumes these instances with property \( \text{parlevel} > 1 \). As a result, multiple signals are processed concurrently by the consuming process.

The \( \text{parlevel} \) property changes the process behavior from synchronous to asynchronous. By default, processes behave synchronously: the next firing can be initiated only after signals from the previous firing have been emitted by the process. However, in a process with \( \text{parlevel} > 1 \) invocations are asynchronous (up to the specified limit): as soon as the function has been invoked, the process is ready for next firing.

The second way is to utilize signal quantities (see section V-B) to have a process consume multiple signals and submit multiple jobs to an external execution management service. Sometimes this may be preferable to avoid the overhead of submitting jobs separately [19].

**Pipelining.** Pipelining is a form of task parallelism analogous to a production line. In terminology proposed by [19], HyperFlow supports blocking, buffered, and superscalar semantics of pipelined parallelism, as shown in Fig. 6.

**Ordering of results.** In a process with \( \text{parlevel} \) property, the order of output signals may not match the order of their input counterparts. Sometimes this is acceptable, for example, when the result is a collection of files. However, whenever the results order needs to be preserved, the \( \text{ordering: true} \) property should be added to the process.

**VI. Workflow examples**

This section presents four workflow examples implemented with HyperFlow. For the first three examples, we intentionally chose workflows from three existing workflow systems: Taverna, Kepler, and Pegasus, in order to compare the implementations and highlight the features of HyperFlow.

The workflows presented in this section are available as examples in the HyperFlow distribution.

**A. Biological pathways workflow: eliminating shim nodes**

The first example is based on a Taverna 2 workflow which, given a Unigene gene identifier, returns the information about biological pathways the gene is involved in. The original Taverna implementation is available through myexperiment.org.

---

3http://github.com/dice-cyfronet/hyperflow
4http://www.myexperiment.org/workflows/2673
The workflow is implemented on the basis of five invocations of the KEGG REST API.\textsuperscript{5} The original Taverna implementation consists of 13 nodes, of which only five are the core domain components, while the remaining eight are beanshell-based shim nodes performing very simple operations related to splitting collections into individual items and various string/XML manipulations.

The HyperFlow implementation, shown in Fig. 7, consists of only five processes. The respective activities are implemented as five JavaScript functions, each of which contains about 15-20 lines of code which invoke the appropriate REST service, using one of several available node.js HTTP client libraries, and performs necessary transformations. Overall the workflow implementation comprises 50 lines of JSON describing the workflow structure, and 100 lines of JavaScript implementing workflow activities. In comparison, the total amount of beanshell glue code required in the Taverna implementation is alone close to 100 lines, which is the result of wrapping small pieces of code as individual nodes (which requires some boilerplate code), and the verbosity of beanshell itself.

In the HyperFlow implementation the glue code is hidden in the JavaScript code, so the workflow graph can be significantly reduced. The invocation of the actual workflow components (REST services) is also hidden in the implementation, so in the case the interfaces change (say, from REST to SOAP), the workflow structure will stay the same, only the internal implementation of functions will be affected. On the contrary, fine-grained shim nodes which extract data produced by services are very sensitive to interface changes. In such a case, the entire structure of the workflow would need to be redesigned.

Let us note that in the above example, we could just as well have chosen to implement explicit shim nodes, exactly replicating the structure of the Taverna implementation. The capabilities of the low-level programming just gives us more flexibility, allowing to choose trade-offs, without enforcing any particular solution or the granularity of workflow nodes.

Finally, let us emphasize that hiding the glue code in the implementation of workflow activities does not reduce the reusability of workflow components. On the contrary, the reusable code can easily be factored out and invoked from diverse workflow activities in various workflows, where only a small amount of glue code would be different each time.

B. Comet workflow: parallel processing of nested data collections

The second example is the Comet workflow, originally implemented in Kepler\textsuperscript{20}, which consumes humidity data from meteorological stations, calculates its statistics (min, max and growing degree days), and generates graphs by invoking the R tool. Again, similarly as in the Taverna example, the original Kepler implementation (based on the Process Networks Director) contains 15 processing nodes the bulk of which are shim nodes.

In an effort to simplify the workflow model, a new Kepler director – Comad – has been implemented\textsuperscript{20}, which is dedicated to processing of XML data collections. The resulting Comad-based workflow has only five processing nodes. The HyperFlow workflow model is equally simple (five processes). However, the implementation itself is significantly easier: it only required 60 lines of the JSON description and 150 lines of the JavaScript implementation of functions.

By contrast, implementing a new Kepler Director is a considerable effort in the first place. The implementation of Kepler Actors also requires much ‘boilerplate’ code and configuration files, whereas HyperFlow functions contain only ‘business logic’, the only HyperFlow-specific code being the input and output data structures (signal arrays). Additionally, the implementation of functions in the HyperFlow workflow utilized only four standard and widely known libraries (such as xpath and DOM parser), familiar to many.

\[\text{KEGG API, http://www.kegg.jp/kegg/rest}\]
expert programmers, who – when using the Comad director – would need to read its user manual first.

Finally, the Comad director has some limitations: it requires a unified data format for representing collections, and supports only pipelining parallelism. In HyperFlow, on the other hand, a new data format can be adopted simply by switching the implementation of the ReadDataSets process’ function. Also, both pipelining and data parallelism are supported (see section V-D). For example, the PlotGraphs process with parameter `parlevel:4` will invoke up to four simultaneous instances of the R tool, so that output graphs will be generated in parallel, significantly reducing the total execution time.

C. Montage workflow: dispatching workflow tasks to the Cloud

This example shows the HyperFlow implementation of a large-scale resource-intensive workflow – the well-known Montage [21]. Montage processes and assembles thousands of small images in order to produce very large, high-quality pictures of celestial objects.

Listing 1: Montage scientific workflow expressed in the HyperFlow language (fragment)

```json
{
  "name": "Montage_10k",
  "functions": [
    "name": "amqp.command", // sends a task to the cloud
    "module": "functions"
  ]
}

"processes": [
  "name": "mProjectPP",
  "config": [
    "executable": "mProjectPP",
    "args": "... "
  ]
],

"ins": [ 0, 3 ],
"outs": [ 1, 2 ]
,
// ... about 10k more processes ...
{
  "name": "mJPEG", // last task: generation of the final image
  "function": "amqp.command",
  "config": [
    "executable": "mJPEG",
    "args": "-ct 1 -gray
shrunken.20090720.143653.22436.fits
-1
.5s
60s
gaussian
-out
shrunken.20090720.143653.22436.jpg"
  ]
],

"ins": [ 22960 ],
"outs": [ 22961 ]
]
,
"signals": [
  "name": "mass-atlas-980529s-j0158174.fits"
],
// ... about 23k more signals
]
,
"ins": [ ... ],
"outs": [ ... ]
}
```

The original workflow has been implemented in the Pegasus workflow system [7]. A Montage workflow contains at least thousands of nodes, so we decided to reuse the Pegasus workflow generator and created a translator which transforms the original Pegasus representation – DAX, directed acyclic graph represented in XML – to the HyperFlow model and description. In such a way we could also make sure that the original workflow structure from Pegasus has been precisely replicated in HyperFlow and the workflows are comparable.

Listing 1 presents a fragment of the HyperFlow description of a small Montage workflow: in this case, over 10,000 processes and nearly 23,000 signals which represent input, output and intermediate files. Each Montage task has been modeled as a separate dataflow process which fires only once. All processes invoke the same function: `amqp.command` which uses a message queue to submit tasks and wait for notifications of their completion. The tasks are picked up by Montage workers deployed in the cloud.

This example demonstrates the independence of the HyperFlow workflow model from the underlying execution environment. We successfully applied exactly the same workflow description to run HyperFlow-controlled Montage on a local laptop, private IaaS cloud using NFS for storage of intermediate files, and Amazon EC2 using S3. In each case, only the function implementation was different. In such a way, the additional layer of abstraction provided by functions fosters workflow engineering, in particular its testability. The workflow model is basically always executable. Only by switching the implementations of functions, we adapt the workflow to a particular execution environment.

D. Flood threat assessment workflow

The final example is a flood threat assessment workflow, originally implemented using HyperFlow. This example is based on our work done within the project ISMOP: a computerized levee monitoring and threat assessment system.6 The research conducted within the ISMOP project spans various aspects including the construction of an artificial levee [22], design of wireless sensors for levee monitoring, development of a sensor communication infrastructure [23], levee modeling and simulation, and a decision support system. HyperFlow is a key component of the ISMOP execution management platform.

One of the main steps in the flood management decision support process is the flood threat assessment workflow shown in Fig. 9. This use case is designed for a typical flood scenario in Poland wherein many kilometers of levees are exposed to a passing flood wave. A high water level lasting for many days can ultimately lead to the loss of stability of the levee structure and a breach [24]. The goal of the workflow is to assess the current stability of levees, partitioned into 10-meter sections. The assessment is based on a large database of pre-computed scenarios simulating

6http://www.ismop.edu.pl
the levee behavior for various initial conditions (more details can be found in [25]). Each process of the workflow computes the ranking of best matching scenarios for a single levee section. The best match is computed by comparing the current sensor readings from a levee section (represented by a RealData signal) with those simulated in the scenarios (the SimulatedScenarios signal). The more dangerous scenarios are selected as best matches for a given levee section, the higher the current threat level is for that section.

We are currently developing a decision support system in which the user will be able to select an arbitrary set of levees for threat assessment and specify a deadline for getting the results. The ISMOP execution management platform will automatically generate an appropriate HyperFlow workflow and create its execution plan in order to meet the deadline [26]. The execution plan will, amongst others, determine the number of required virtual machine instances where the ComputeScenarioRanks jobs will be computed.

VII. Conclusion

We presented HyperFlow: an innovative model of computation, programming language, and enactment engine for scientific workflows. The novelty of HyperFlow lies in increasing workflow programming productivity which is achieved with a number of key features. The first one is a simple programming model, yet based on a formal model of computation, sufficiently expressive for complex workflow patterns. The second one is the choice of a mainstream programming language (JavaScript + JSON) and node.js as the foundations of the workflow programming and runtime environments. Consequently, the workflow developer gains access to a rich ecosystem encompassing large developer community, over 84,000 of reusable software packages, and rich learning resources.

Combining a well-defined workflow model with low-level generic scripting solves some well-known problems associated with visual workflow programming. First, it facilitates hiding of the glue code inside functions (JavaScript code), basically eliminating shim nodes. Second, it also eliminates the boilerplate code normally associated with adapting new processing components to the runtime environment of a workflow system. Third, the invocation of processing components is independent of the HyperFlow engine runtime, fostering their reusability.

HyperFlow is currently being integrated as a key workflow engine in several projects. In the ISMOP project\(^7\), as described in section VI-D, HyperFlow is a crucial part of the execution management platform for the ISMOP flood monitoring and threat assessment system. Within the PL-Grid Core project\(^8\), HyperFlow is adapted as part of a platform for the execution of scientific applications on the PL-Grid Cloud infrastructure. In the EU FP7 funded project PaaSage\(^9\), HyperFlow is adapted as part of the solution to enable execution of multi-cloud large-scale scientific applications in the PaaSage Cloud platform.

Acknowledgment

This work is partially supported by the National Centre for Research and Development (NCBiR), Poland, project PBS1/B9/18/2013 – (no 180535); and by AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, statutory project no. 11.11.230.124.

References


\(^7\)http://www.ismop.edu.pl
\(^8\)http://www.plgrid.pl
\(^9\)http://www.paaasage.eu


