Future Generation Computer Systems 55 (2016) 147–162

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows

Bartosz Balis

AGH University of Science and Technology, Department of Computer Science, Krakow, Poland

HIGHLIGHTS

- A model of computation and system for scientific workflows, HyperFlow, is proposed.
- HyperFlow aims at high development productivity of skilled programmers.
- The HyperFlow Model of Computation combines simplicity with high expressiveness.
- Complex workflow patterns can be implemented using a simple syntax.
- HyperFlow enables a fully distributed and decentralized workflow execution.

ARTICLE INFO

Article history:
Received 30 March 2015
Received in revised form
13 August 2015
Accepted 28 August 2015
Available online 26 September 2015

Keywords:
Scientific workflows
Process networks
Workflow programming
Workflow patterns
Workflow enactment

ABSTRACT

This paper presents HyperFlow: a model of computation, programming approach and enactment engine for scientific workflows. Workflow programming in HyperFlow combines a simple declarative description of the workflow structure with low-level implementation of workflow activities in a mainstream scripting language. The aim of this approach is to increase the programming productivity of workflow developers who are skilled programmers and desire a programming experience similar to the one offered by a mature programming ecosystem. Combining a declarative description with low-level programming enables elimination of shim nodes from the workflow graph, considerably simplifying workflow implementations. The workflow description is based on a formal model of computation (Process Networks) and is characterized by a simple and concise syntax, utilizing just three key abstractions—processes, signals and functions. Yet it is sufficient for expressing complex workflow patterns in a simple way. The adopted model of computation implemented in the HyperFlow workflow engine enables fully distributed and decentralized workflow enactment. The paper describes HyperFlow from the perspective of its workflow programming capabilities, the adopted model of computation, as well as the enactment engine, in particular its distributed workflow enactment capability. The provenance model and logging features are also presented. Several workflow examples derived from other workflow systems and reimplemented in HyperFlow are extensively discussed.

1. Introduction

Workflows have been widely adopted as a programming model for scientific applications. Structuring a scientific application as a workflow provides a convenient level of abstraction for expressing a scientific problem, shifts the complexity of parallel processing and its optimization to a workflow management system and facilitates provenance tracking and reproducibility [1,2].

Workflow programming is a crucial step in the scientific workflow lifecycle in which the workflow developer composes a workflow graph, i.e. specifies the workflow activities and their dependencies. The second aspect of workflow programming, which – contrary to graph composition – has not received sufficient attention in the literature, is the programming of workflow activities themselves. The majority of existing workflow development systems are aimed at domain scientists and attempt to hide the complexity related to low-level programming [1]. Consequently, the prevalent approach in such systems is that the user is given a predefined 'palette' of components which can be composed into a workflow. Adding new components to the palette is not straightforward, requires using proprietary mechanisms of a given workflow system, and cannot usually be done by the workflow developer (domain scientist).

This approach, typically based on visual programming, is perfectly valid for domain scientists with limited programming

E-mail address: balis@agh.edu.pl.

http://dx.doi.org/10.1016/j.future.2015.08.015
0167-739X/

skills and can be particularly effective for systems focusing on a particular discipline where workflows are relatively small and many reusable workflow components are already available, a representative example being the Galaxy workbench [3]. However, in many cases workflow development needs to involve experienced programmers. For example, workflows can be too large and complex to be composed by the scientists who instead interact with specialized science gateways where workflow instances are automatically generated [4]. A similar case occurs when workflow components need to be developed from scratch. In such cases, a workflow programming approach is desired which provides the programming capabilities similar to those of a mainstream programming language without forfeiting the advantages of a formal workflow model.

This paper presents HyperFlow: a model of computation, programming approach and enactment engine for complex distributed workflows. The novel contribution of HyperFlow concerns three areas: workflow modeling, programming, and execution.

1. Simple and expressive model of computation. HyperFlow provides a model of computation for workflows which combines simplicity with broad expressiveness. The model is inspired by Process Networks and is based on only three abstractions: processes, signals, and functions. Despite its simplicity, the model enables advanced programming capabilities and a rich set of workflow patterns that can be expressed with it, including patterns for processing of data collections, patterns for parallel processing, and various data and control flow patterns.

2. Innovative workflow programming approach. In HyperFlow, a workflow implementation involves a simple declarative description of the workflow graph, expressed in a JSON syntax, and implementations of workflow activities as JavaScript/`Node.js` functions. This allows workflow developers to preserve all the advantages of a graph model of parallel computation while leveraging benefits of low-level programming in a mainstream general-purpose language and its runtime environment. These benefits include the simplification of workflow implementations (e.g. by handling glue code in the low-level implementations of workflow activities which enables elimination of shim nodes), and gaining access to a mature programming ecosystem with many developers, reusable software packages and rich learning resources.

3. Lightweight workflow runtime environment. Finally, HyperFlow provides a workflow enactment engine, implemented as a lightweight `Node.js` [5] application. The engine implements the HyperFlow model of computation end enacts the workflow, but the actual execution of workflow activities only requires the `Node.js` runtime environment which makes their implementation reusable outside the HyperFlow context. The workflow execution model of the HyperFlow engine is inherently distributed and decentralized. The engine provides a REST API which enables remote workflow management and delivers the capability to execute workflows in a fully distributed and decentralized manner where multiple engine instances cooperatively enact a workflow, without centralized control.[1]

This paper is a substantial extension of the previously published conference paper [6]. Section 2 overviews related work. Section 3 extensively describes the HyperFlow workflow model, programming approach and various programming capabilities, including the implementation of representative complex workflow patterns. Section 4 discusses the model of computation implemented in HyperFlow as well as its provenance model. In Section 5, the HyperFlow enactment engine is presented. Section 6 studies example workflows implemented in HyperFlow. Finally, Section 7 concludes the paper.

2. Related work

Two main approaches to workflow programming in existing workflow systems are the ones based on, respectively, visual or textual graph composition, and custom-designed dataflow scripting languages. In the former approach, the developer composes the workflow by selecting workflow activities from the available palette of components, configuring them, and connecting their inputs and outputs. The components in the palette can represent particular domain-specific procedures related to a specific discipline (life sciences, astronomy, Earth sciences, etc.), but they can also provide control flow constructs (e.g. loops and conditional statements), data manipulation operations (e.g. string processing, array operations and data transformations), or other useful high-level functions (HTTP or Web Service client, file system operations, math functions, etc.). Many workflow systems follow this general approach while using different names for the workflow components, for example *activities* in ASKALON [7], *actors* in Kepler [8], *services* in Taverna [9], and *tools* in Triana [10].

Adding new components to the existing palette is usually possible but it requires using proprietary mechanisms of a given workflow system. For example, in ASKALON workflows are composed of abstract activities which are mapped to concrete executable deployments. Consequently, all activities (even simple ones such as those handling data conversions) need to be semantically described and registered in an activity registry. In Kepler, new actors can be programmed in Java by implementing a specific interface in order to conform to the general actor lifecycle. For example, in [11] a set of new actors needed to be implemented in order to support submission and monitoring of jobs from Kepler workflows to a cloud infrastructure. In CLAVIRE [12], software packages need to be described in detail using a domain-specific language *EasyPackage* in order to be invokable from a workflow. In WS-VLAM, workflow activities are implemented as python modules that need to adhere to a specific interface [13]. Such mechanisms, while attempting to unify the way diverse software components are invoked from a workflow, ultimately remain proprietary solutions that the developers are forced to use, resulting in extra software development effort, decreased interoperability, and problems related to software maintenance.

In Taverna, workflows are developed by composing components (called 'services') in a graphical editor. One of the supported service types is a *beanshell* service which encapsulates an arbitrary piece of Java code. Workflow implementations are typically broken down into two types of services:

- reusable *domain services* with stable interfaces; in practice these are implemented as Web services ("Taverna workflows essentially specify Web service compositions" [14]);
- non-reusable *shim*[2] services with ad hoc interfaces that act as adaptors/connectors between "domain" services and are implemented on the basis of beanshell services.

Wrapping glue code as explicit nodes in the workflow graph has a number of disadvantages. Shim nodes introduce noise which obscures workflow presentation and its provenance trace, because the user is not interested in trivial transformations in the scientific pipeline. A solution presented in [15] proposes special annotations provided by the user or the developer in order to distinguish interesting and uninteresting workflow activities. Obviously such

[1] The HyperFlow engine is available as open-source software at https://github.com/dice-cyfronet/hyperflow.

[2] Shim node is a workflow activity which does not perform a scientific procedure, but serves as a connector/glue between other domain-specific workflow activities.

a solution increases development effort and requires the workflow system to support the annotations. In HyperFlow, on the other hand, trivial transformations are hidden in the low-level code, so the workflow graph is focused on the core scientific pipeline. Another problem with shim nodes is that they make the workflow graph very sensitive to changes in the interfaces and/or data formats of the invoked domain services. Such changes require the redesign of the associated shim nodes and may imply major changes in the entire workflow graph (e.g. additional shim nodes). On the contrary, with glue code hidden, the necessary changes affect only the low-level implementation.

A possible option would be to implement a workflow entirely on the basis of beanshell services that besides trivial transformations also invoke domain (Web) services from the beanshell code. However, this approach would entail a number of problems due to Taverna's workflow programming environment and philosophy wherein a workflow is composed of graphical blocks. In a beanshell-only workflow, the developer would not be taking advantage of the rich palette of domain components, because every workflow activity would be a generic beanshell service. Taverna supports workflow debugging at the level of graphical blocks, not the beanshell code, so wrapping calls to domain procedures in beanshell nodes would make the workflow more difficult to debug. In HyperFlow, on the other hand, this does not pose a problem, because the basic unit of workflow composition is a JavaScript function, so any `Node.js` debugger can be used. Also, Taverna's semantic provenance relies on service annotations [14], so hiding invocations of domain services inside the beanshell code would break advanced provenance logging.

Some workflow systems, notably Pegasus [16] and WS-PGRADE [17,18], follow a significantly different approach. In these systems workflow components are constrained to external executable programs. Mapping workflow activities to executables does have a number of benefits. First, they can be implemented in any programming language. Second, stand-alone executable programs are independent of the workflow system runtime environment, and hence conducive to reusability and interoperability between workflow systems. Significantly, the SHIWA project [19], which provides technology to convert between different workflow representations through a common Interoperable Workflow Intermediate Representation (IWIR) [20], only supports this limited workflow execution model. Ensuring a deeper interoperability at the level of fine-grained system-specific workflow components would be much more challenging, if not infeasible. However, the restriction to executable programs has also disadvantages: even small pieces of code have to be wrapped as programs and consequently implement a significant amount of boilerplate code related to reading input and writing output data, parsing configuration, implementing error reporting conventions, etc. For some types of workflows, e.g. those orchestrating web services and exchanging small amounts of XML data it becomes cumbersome.

A number of approaches to programming scientific applications are based on scripting languages. These are typically custom-designed dataflow languages, such as Swift [21] and gscript [22], both of which are based on asynchronous operations and single-assignment future variables. Such an approach provides similar benefits as typical graph-based workflow models: data-driven concurrency semantics, implicit deterministic parallelism, and well-defined data dependencies. In fact, gscript is actually a script-based counterpart of the GWENDIA workflow language [23]. In Swift, processing components are not written in the Swift language, but are invoked as so called *leaf functions* which can be either executable programs or Tcl functions, the native language of the Swift runtime, Turbine. Indirectly, Swift can also invoke C, C++ and Fortran functions which need to be wrapped into Tcl. 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. Invokers, similarly to processing components of workflow systems, are proprietary mechanisms for invoking software components and suffer from the same problems related to development effort, interoperability and software maintenance.

Finally, there exist solutions which are based simply on providing scientific programming abstractions (APIs) for general-purpose programming languages. Baranowski and others [24] propose an approach to programming scientific applications based on Ruby scripts. A high level API (called Grid Operation Invoker) is used for abstracting underlying computations as RPC-like calls [25]. Parallelism is achieved through asynchronous calls and blocking waiting for operation results. Programming in a mainstream general-purpose scripting language solves many problems of workflow programming approaches based only on declarative graph composition or custom-designed scripting languages. Rich programming constructs built into the language make it easy to write glue code between invocations of processing components. The developer gains access to a mature programming ecosystem with many reusable software packages, large community of developers and rich learning resources. However, the lack of a well-defined workflow model and properties associated with it creates serious problems in other areas. Parallel programming becomes more challenging and the desired deterministic parallelism is much more difficult to achieve. Provenance logging, in turn, basically requires the tracking of variable dependencies [24].

The following section describes the HyperFlow approach which combines the benefits of a declarative workflow description and low-level programming in a mainstream programming language.

3. HyperFlow workflow model and programming approach

3.1. Overview of workflow programming in HyperFlow

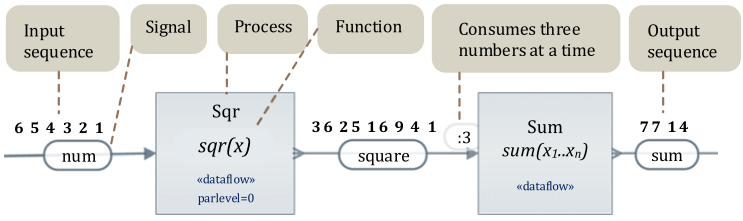
Programming a workflow in HyperFlow involves defining its graph as a JSON data structure, and programming its activities as JavaScript functions. The workflow graph definition comprises an array of **processes** (workflow activities) and an array of **signals** (data exchanged between processes). Each process is assigned a **function** which implements its activity.

For better understanding of the HyperFlow workflow programming approach, let us consider an intentionally simple example of a workflow which consumes a sequence of numbers and, for every three, produces the sum of their squares. Fig. 1 illustrates this workflow, presents its graph model in JSON and the implementation of its activities in JavaScript.

The workflow is composed of two processes named `Sqr` and `Sum`, and three signals named `num`, `square` and `sum`. The main properties of each process are its input and output signals and the associated function, as well as the process type (which determines the general process behavior, described in more detail in Section 3.2).

The `Sqr` process consumes numbers and produces their squares. It does it in parallel owing to the `parlevel:0` property which means that multiple firings of the process will run concurrently (value 0 means unlimited). In such a case results may be produced out of order, so the `ordering` property is added to prevent this. The `Sum` process consumes three squares in each firing – which is specified by the signal quantity modifier `squares:3` – and produces the final sum.

When a process is fired, its function is invoked. The function has four parameters: `ins`—data structure in which the input signals that triggered the process are provided; `outs`—data structure for writing the output signals to be emitted in this firing of the process; `config`—additional configuration data; and `cb`—the callback to be



(a) Sum of squares of three numbers in HyperFlow (illustration).

```
{
    "name": "Wf_SumSquares",
    "processes": [ {
        "name": "Sqr",
        "type": "dataflow",     // process type
        "function": "sqr",      // function to be invoked upon process firing
        "parlevel": 0,          // the squares will be computed in parallel...
        "ordering": "true",     // ...but the results will be ordered
        "ins":  [ "num" ],      // signals consumed by this process
        "outs": [ "square" ]    // signals emitted by this process
    }, {
        "name": "Sum",
        "type": "dataflow",
        "function": "sum",
        "ins":  [ "square:3" ], // 3 signals must be collected to fire the process
        "outs": [ "sum" ]
    } ],
    "signals": [ {
        "name": "num",
        "data": [ 1, 2, 3, 4, 5, 6 ]  // initial signal instances
    }, {
        "name": "square"
    }, {
        "name": "sum"
    }  ],
    "ins": [ "num" ],
    "outs": [ "sum" ]
}
```

```
function sqr(ins, outs, config, cb) {
    var n = Number(ins.num.data[0]);
    outs.square.data = [ n * n ];

    cb(null, outs);
}
```

```
function sum(ins, outs, config, cb) {
    var sum=0.0;
    ins.square.data.forEach(function (n) {
      sum += n;
    });
    outs.sum.data = [ sum ];
    cb(null, outs);
}
```

(b) Workflow graph in JSON and workflow activities implemented in JavaScript.

Fig. 1. Sum of squares workflow: the workflow computes sum of squares of every three numbers that arrive as the 'num' input signal.

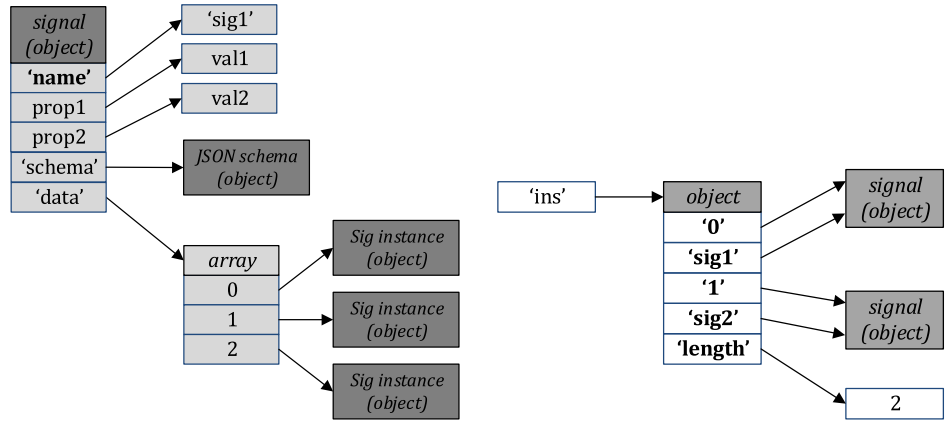
invoked when the activity is finished, via which the output signals are returned. For example, function `sqr` has three lines of code: the first one reads the input data, the second one writes the output data, while the third one invokes the callback to actually return the output data to the engine.

The data structure representing a signal is shown in Fig. 2(a). A signal object contains both metadata and data, i.e. one or more signal instances, stored in the `data` array. The function's `ins` and `outs` are simply arrays of signals, as shown in Fig. 2(b).[3] The data structures for writing output signals initially contain only metadata. The function is supposed to store the actual data (instances of the output signals) in the `outs[i].data` array and invoke the callback, as shown in Fig. 1(b). If the function returns no `data` arrays, no signals will be emitted which is the simplest way to implement filtering. Let us note that signals are not meant for transfer of large data payloads. Typically they will contain links to actual data sets (e.g. a file path or an FTP URL), so that data transfers between workflow activities can leverage efficient communication channels.

The initial input signals for the workflow can be sent after its instance has been created, e.g. using the REST API of the HyperFlow engine (see Section 5). However, they can also be provided directly in the workflow description in which case they will be emitted automatically upon creation of the workflow instance. In the sum-of-squares example six initial instances of the `num` signal are specified (Fig. 1). Consequently, when the workflow is executed, it immediately produces two outputs: `14` and `77`.

HyperFlow does not impose any data type system: signals are just arbitrary JSON structures. However, signal metadata provides means to adopt domain-specific data types in a particular workflow. Metadata may contain a `schema` object compliant with

[3] Both `ins` and `outs` arrays are actually JavaScript Array-Objects, so they conveniently can be treated either as arrays (signals can be referenced by index, e.g. `ins[0]`) or objects (signals can be referenced by name, e.g. `ins.sig1`).



(a) Structure of a signal (example).

(b) Structure of function inputs/outputs (example). Signals can be referenced by their index or name.

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. The input signals contain both metadata and data (signal instances), the output signals only metadata.

the JSON schema.[4] If a schema is provided, signal instances must conform to it and will be validated by the HyperFlow engine.

3.2. Process types

A key concept of the HyperFlow workflow model is the **process type** which determines the general behavior of a process. Implementing new process types is also a powerful way of extending the capabilities of the model while preserving full backward compatibility. Currently the following process types exist, which have been gradually added to the model over the course of its development: *dataflow*, *choice*, *foreach*, and *join*.

Dataflow. A `dataflow` process simply waits for signals on *all* its inputs, invokes the function, and emits signals to all its outputs. This is the default process behavior.

Foreach. Unlike `dataflow,` a `foreach` process is triggered by *any one* of its inputs, invokes the function, and emits a signal to the corresponding output. Consequently, the number of outputs must be equal to the number of inputs.

Choice. A `choice` process is similar to `dataflow,` the only difference being that in a given firing it may emit only a subset of its output signals. The decision on which signals should be emitted is made in the process' function which simply returns signal instances for only those signals which should be emitted. Such a semantics enables a variety of workflow patterns including conditional execution, multi-choice, deferred choice, filtering, and data routing.

Join. A `join` process synchronizes parallel branches of execution represented by its input signals. The `join` process type has been designed to enable the implementation of various "*n*-out-of-*m*" patterns, e.g. *discriminator* and *structured join* (described in Section 3.6). In scientific applications these patterns are used, e.g., to wait for results from *n* jobs out of submitted *m*, for example in Monte Carlo simulations [26]. Two parameters control the precise behavior of a `join` process in a given firing:

- N_b (active branches count): how many inputs out of total N ($N_b \leq N$) will be active (i.e. will receive signals) in this firing?
- N_j (join count): how many activated inputs are sufficient to fire the process? ($N_j \leq N_b$).

For example, in a `join` process with 4 inputs, parameters $N_b = 3$, $N_j = 1$ denote the following behavior:

1. Three (N_b) out of four inputs will be active in this firing, i.e. will receive signals.
2. The process will fire as long as there is data on any one (N_j) input (1-out-of-3).
3. After firing, the process will wait for data on two remaining active inputs, discard them, and reset (go back to step 1).

3.3. Control signals

The HyperFlow model features so called *control signals* which are not passed to the function, but they affect the process behavior in another way, depending on the type of the control signal. A signal with property `control: <type>` is a control signal of a given type. The following control signals are currently available in the HyperFlow model:

Next. If added as an input of a process, the `next` signal is required for firing alongside other data signals. If added as an output of a process, the `next` signal will be emitted automatically after each firing of the process. Such a semantics can be used to enforce synchronous execution of any workflow subgraph and can be used to implement loop-like constructs. See discussion on data parallelism in Section 3.5 and Fig. 6 for an illustration.

Done. If added as an input of a process, the arrival of the `done` signal will cause this process to terminate right after the current firing has been finished. If added as an output, it will be emitted as the last signal prior to the termination of the process.

Count. The `count` signal is always associated with another data signal and is used to dynamically set the signal quantity of this signal, which is particularly useful when producing and consuming data collections—see Section 3.4 for an illustration. When used on input, the `count` signal determines the number of instances of the associated data signal which will be required for the next firing of the process. When used on output, the `count` signal is automatically emitted alongside the associated data signal and denotes the number of instances of this signal just emitted in this firing.

Merge. The `merge` signal can only be produced by a `choice` process and consumed by a `join` process. This signal denotes the number of branches activated by a `choice` process in a given firing, and the number of branches which need to be activated to fire the associated `join` process. Section 3.6 illustrates the use of the `merge` signal for the implementation of complex workflow patterns.

[4] http://json-schema.org.

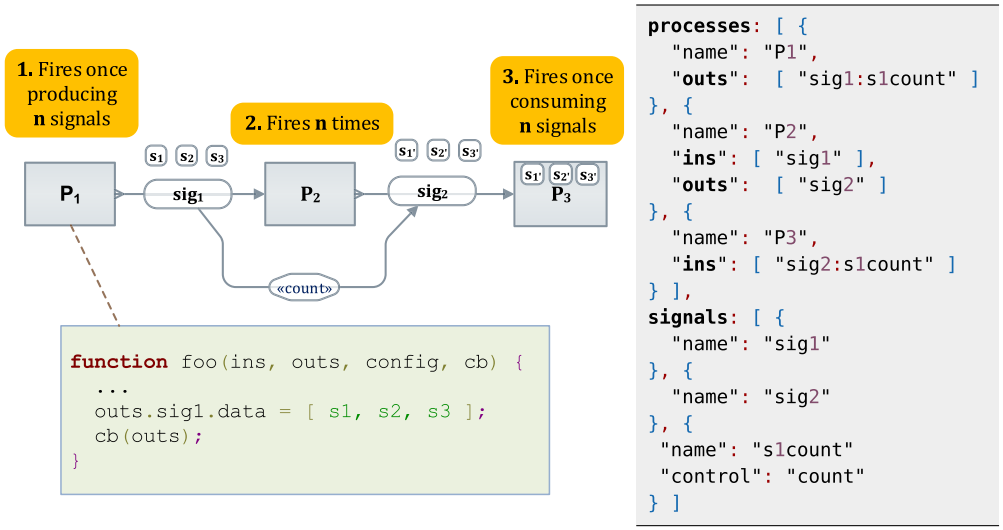


Fig. 3. Producing and consuming data collections whose size is known at runtime. Process P1 produces a collection of signals simply by returning an array of elements in its function. Each element in the collection is processed by P2. Finally, P3 needs to collect all results (sig2) as a collection of signals. The number of elements in the collection is passed from P1 to P3 dynamically via the s1count control signal. To this end, the output of P1 and the input of P3 are related with the s1count tag.

3.4. Data collections

In HyperFlow, a collection of data elements is represented simply as a sequence of signals. A process can *produce* multiple signals (a collection) in a single firing simply by returning an array of signal instances. A process can also *consume* a collection of signals by specifying a *signal quantity* for a given input signal using the following syntax: `ins: [ "<name>:<quantity>" ].` In the simplest case, `quantity` is a number denoting how many signals are required for firing, for example `square:3.`

Sometimes, however, the quantity of signals we need to consume varies from firing to firing and can only be determined at runtime. In this case, `quantity` can be a tag, as shown in Fig. 3. This figure illustrates a typical workflow pattern: process `P1` generates a collection of signals which are processed one by one by `P2.` Process `P3,` in turn, needs to collect all results produced by `P2` before the workflow can continue. Process `P3` knows how many elements need to be collected thanks to the `s1count` tag which relates the output of `P1` with the input of `P3.` Tag `s1count` is actually the name of a `count` control signal which contains the information of the number of signals produced in a given firing, and is automatically emitted from process `P1` to process `P3` (see also Section 3.3).

3.5. Parallelism

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

Task parallelism. Task parallelism, which simply means different tasks running in parallel, is implicit in HyperFlow: a process is fired as soon as its input signals are available.

Data parallelism. Data parallelism means multiple instances of the same task running in parallel for different data elements. HyperFlow provides two idioms for this type of parallelism, illustrated in Fig. 4. The first one involves a process producing a collection of signals (see Section 3.4), and another process consuming the elements of the collection while having property `parlevel` set to value >1.

The way the `parlevel` property affects the process behavior is shown in Fig. 5. By default, process firings are synchronous: the next firing is executed only after signals from the previous one have been emitted. With `parlevel>1` (or 0) this behavior is changed to asynchronous: the process is ready for a next firing as soon as the function from the previous firing has been invoked, without waiting for the callback. The value of the `parlevel` property denotes how many firings can run concurrently. Concurrent firings may produce the output signals out of order with respect to the input counterparts. If this is not desirable, the `ordering` property can be set to preserve the ordering.

Pipelining. Pipelining is a form of task parallelism where multiple stages of a processing pipeline run concurrently, similarly to a production line. Using the terminology proposed in [27], HyperFlow supports blocking, buffered, and superscalar semantics of pipelined parallelism, as explained in Fig. 6.

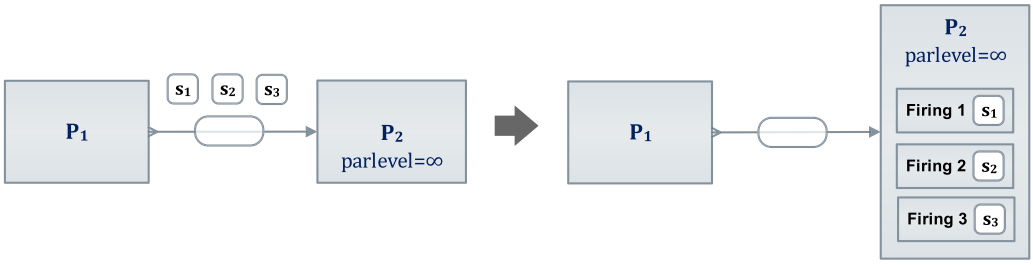
3.6. Support for workflow patterns

To conclude the overview of HyperFlow workflow programming capabilities, let us consider its support for implementing representative workflow patterns found on http://www.workflowpatterns.com. This well-known web page describes in detail over 80 different patterns. Here, for the sake of brevity, we do not include the full description of the discussed workflow patterns, just describing their implementation in HyperFlow. We mostly focus on control flow patterns as of all patterns described on the page, these are most relevant to scientific workflows. Most data patterns found there, on the other hand, are quite specific to business workflow management systems. For example, shared variables are rare in scientific workflow systems which are dataflow-oriented and all state is exchanged through messages between workflow activities.

Basic Control Flow Patterns – such as *Sequence, Parallel Split*, and *Synchronization* – are trivially implemented in HyperFlow, simply by connecting processes with signals in a desired configuration. Simple conditional execution patterns: *Exclusive Choice* and *Multi Choice* are directly supported by the `choice` process type.

Data routing. This pattern can be implemented using a `choice` process whose function evaluates a condition based on the values of the input signals and returns only selected output signals, depending on the outcome of the evaluation.

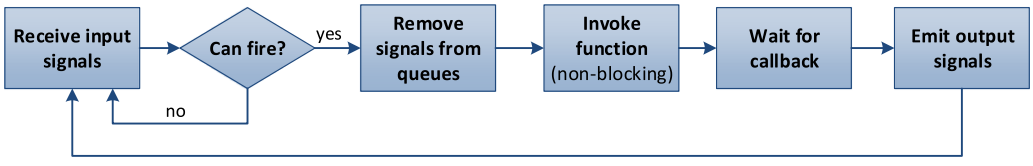
Simple Merge and *Multi Merge*. These patterns can be implemented using the multiple-sources-single-sink configuration where multiple processes emit the same signal to another single process.



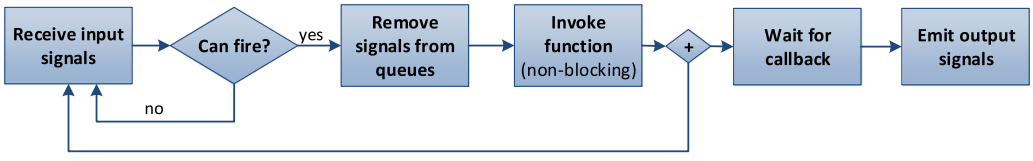
(a) Data parallelism using multiple firings. P_1 produces a collection of signals which are processed by P_2 in multiple concurrent firings. The ordering of results may be preserved by setting the `ordering` property.

(b) Data parallelism through consuming a collection of signals and starting multiple jobs via an external execution service.

Fig. 4. Two idioms for data parallelism in HyperFlow.



(a) Synchronous process firing.



(b) Asynchronous process firing.

Fig. 5. Workflow execution with synchronous versus asynchronous process firings.

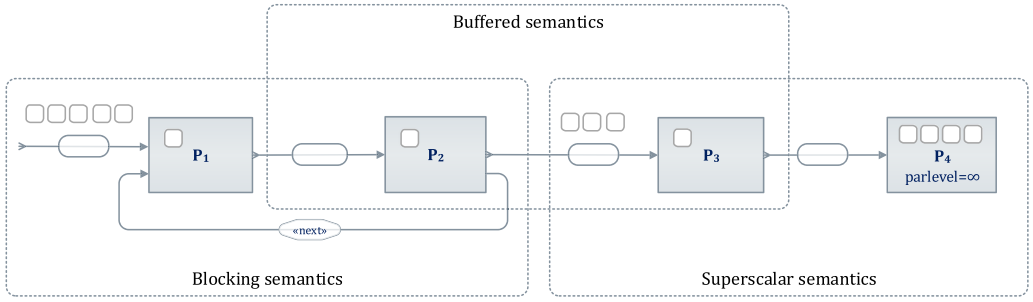


Fig. 6. Three semantics of pipelined parallelism supported by HyperFlow. **Blocking**: due to the `next` signal, process P1 is blocked by P2, so that P1 will produce a signal only when P2 is ready to consume it. **Buffered**: signals produced by P2 are buffered until P3 is ready to consume them. **Superscalar**: P4 processes multiple available signals immediately in multiple concurrent firings.

Generalized AND-Join. A dataflow process naturally fulfills this pattern owing to the mechanism of queuing input signals built into the HyperFlow model.

Thread Split. This pattern is enabled by a process with the `parlevel` property. Multiple signals simultaneously arriving at such a process are processed in concurrent firings ("threads"). The value of the `parlevel` property indicates how many such firings can be executed concurrently.

Thread Merge. In HyperFlow, this pattern can be interpreted as follows: a process should wait for a given number of signal instances on a designated input before passing control to the next process. There are two ways to implement this pattern. The first one uses the mechanism of dynamic signal quantity, as shown in Fig. 3, where process P_3, in addition to consuming a collection of signals, also merges multiple firings triggered by this collection. This implementation can only be used when the signals to be collected are derived from a collection of elements. If this is not the case, the pattern alternatively can be implemented as shown in Fig. 9(a). There, process P_0 has two input signals: `cnt` represents the number of signals to be collected (i.e., threads yet to be merged), while `elem` denotes the signals themselves. The output signal `list` represents passing control to the next process. In this case, only when four `elem` signals have been collected, the `list` signal is emitted.

The next five patterns can be implemented using the `join` process type (described in Section 3.2). Four of them are illustrated in Fig. 7.

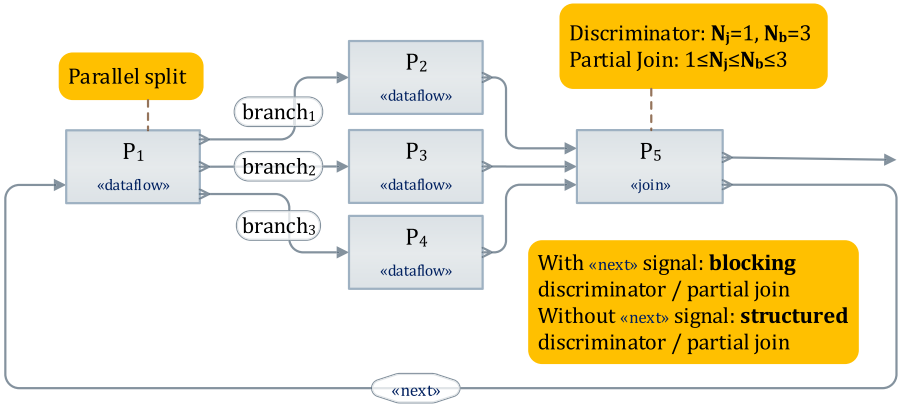


Fig. 7. Illustration of workflow patterns enabled by the `Join` process type: blocking discriminator and blocking partial join (with 'next' signal); structured discriminator and structured partial join (without 'next' signal). When the 'next' signal is present, the next firing of P1 can only start after the current firing of P5 has finished.

Structured Discriminator. A `join` process with parameters $N_j = 1$, $N_b =$ # of inputs, implements this pattern.

Blocking Discriminator. The implementation of this pattern is the same as *structured discriminator* with an additional `next` signal emitted by the `join` process after reset.

Structured Partial Join. The `join` process type is a direct implementation of this pattern, where N_j denotes the number of branches to join and N_b denotes the number of active branches.

Blocking Partial Join. The implementation of this pattern is the same as *Structured Partial Join* with an additional `next` signal emitted by the `join` process after reset.

Local Synchronizing Merge. This pattern can be implemented using the `choice` and `join` processes connected with the `merge` control signal. This signal sets the N_b and N_j parameters of `join` to the number of branches activated by the `choice` process in a given firing.

A more complete and continuously updated description of workflow patterns implemented in terms of HyperFlow model can be found at the HyperFlow engine wiki pages.[5]

4. Model of computation and provenance

4.1. HyperFlow model of computation

The HyperFlow model of computation is similar to, and inspired by Kahn process networks [28], as well as the related model—dataflow process networks [29]. In Kahn's model, processes communicate by passing data objects (called tokens) through unidirectional unbounded FIFO channels using blocking reads and non-blocking writes. A process is allowed to block only on one input channel at a time. Furthermore, each token is produced and consumed exactly once, which means that channels are strictly point-to-point.

In HyperFlow, a running workflow instance is essentially a collection of processes, each of which runs autonomously according to the following cycle: (i) *receiving input signals* and queuing them for later processing if they cannot be consumed immediately; (ii) *testing if a firing pattern is fulfilled*, i.e. all required input signals have arrived; (iii) *invoking the activity function*: the signals that fired the process are removed from the queues and passed to the function; (iv) *awaiting for outputs* which are returned by the function's callback; (v) *emitting outputs as signals* to their sinks.

$$F^f_{dataflow}(s_{in_1},\ldots,s_{in_n}) = f(s_{in_1},\ldots,s_{in_n}) = (s_{out_1},\ldots,s_{out_m})$$
$$\text{where } s_{in_*} \neq \perp \quad s_{out_*} \neq \perp$$
$$F^f_{foreach}(s_{in_1},\ldots,s_{in_n}) = (f(s_{in_1}),\ldots,f(s_{in_n})) = (s_{out_1},\ldots,s_{out_n})$$
$$\text{where } s_{in_*} \neq \perp$$
$$F^f_{choice}(s_{in_1},\ldots,s_{in_n}) = f(s_{in_1},\ldots,s_{in_n}) = (s_{out_1},\ldots,s_{out_m})$$
$$\text{where } s_{in_*} \neq \perp$$
$$F^f_{join}(s_{in_1},\ldots,s_{in_n},N_b,N_j) = f(s_{in_{i_1}},\ldots,s_{in_{i_{N_j}}}) = (s_{out_1},\ldots,s_{out_m})$$
$$\text{where } n \geq N_b \geq N_j \quad s_{in_{i_1}},\ldots,s_{in_{i_{N_b}}} \neq \perp \quad s_{in_{i_{N_{b+1}}}},\ldots,s_{in_{i_n}} = \perp$$

Fig. 8. HyperFlow processes can be described as functions F from input signals to output signals. If the activity function f is a pure function, then F also is a pure function, and consequently the related process is continuous (preserves determinacy).

One of the most important properties of Kahn Process Networks is determinacy. Deterministic parallelism with explicit non-deterministic constructs is argued to be the desirable model of parallelism [30]. A process network is *determinate* if given specific input sequences, all other sequences are determined. Kahn proved that the network is determinate as long as its processes are *continuous functions* from input sequences to output sequences. Intuitively, a function is continuous if the output tokens are purely a function of the input tokens, and more input can only yield more output. A more formal definition can be found in [29]. Fig. 8 shows that HyperFlow processes can be expressed as functions F from input signals to output signals. Note that function F is not identical with the function f implementing the process activity: it also depends on the type of the process.

Determinacy of HyperFlow processes can also by analyzed by means of the *firing rules* formalism used in dataflow networks. A firing rule is a set of patterns, one for each of n inputs of a process [29]: $R = \{R_1, R_2, \ldots, R_n\}$. Each pattern specifies the minimal number of tokens that must be present (and will be consumed) on each input for the process to fire. For example the following rule: $\{[*], [*, *], [1], \perp\}$ denotes a single token (of any value) on input 1, any two tokens on input 2, token with value 1 on input 3, and no tokens on input 4 ($\perp$ denotes an empty sequence).

For a dataflow process to be continuous, it is sufficient that (1) firings are functional, i.e. the output tokens are purely a function of input tokens; (2) the set of firing rules is *sequential* which means that they can be effectively tested in some pre-defined order using only blocking reads on individual input ports. For example, in HyperFlow both *dataflow* and *choice* processes have only one firing rule: $R_{dataflow} = R_{choice} = \{[*], [*], \ldots, [*]\}$. In both cases the rule can be tested with blocking reads on individual inputs, in any order, therefore the rule is sequential. This leads to the conclusion that *dataflow* and *choice* processes are continuous as

[5] https://github.com/dice-cyfronet/hyperflow/wiki/Workflow-patterns.

Table 1
Extended '*Read – Write – State change*' provenance model.

Event type	Meaning
Read	A signal was read by a process and added to its state.
Write	A signal was written by a process.
State-reset	All signals added in previous firings are removed from the process' state.
State-remove	A specific signal is removed from the process state.

long as their function preserves the purely functional relationship between input and output signals.

The *foreach* process, in turn, has n firing rules, one for each input: $R_{foreach,1} = \{[*], \perp, \ldots, \perp\}, \ldots, R_{foreach,n} = \{\perp, \ldots, \perp, [*]\}$. Such a set of firing rules fails the procedure for identifying sequential rules given in [29, p. 17]. However, we can note that a foreach process with n inputs and outputs, can be replaced in a workflow graph with n dataflow processes, each with one input and output, and the same function. After such a transformation, it is clear that a *foreach* process is also continuous under the same conditions as *dataflow*. Other process types can easily be analyzed in a similar manner.

The HyperFlow model relaxes some constraints of the original Kahn Process Networks which in certain well-defined cases may lead to non-determinism. These relaxed constraints occur in the following situations:

- In HyperFlow a signal may have multiple sources and a single sink. Such a configuration violates the rule that *multiple processes may not write to the same channel*, but it also enables such workflow patterns as Simple Merge and Multi Merge.
- The `parlevel` property also leads to non-determinism because output signals may be produced in a random order, unless the `ordering` property is set.

Note that another rule for Process Networks that *many processes may not read from the same channel* is always preserved, even though a single output signal can be connected to multiple processes. However, in such a case HyperFlow simply implicitly produces multiple copies of any signal, which does not cause non-determinism.

4.2. Provenance model

The provenance model adopted in HyperFlow is an adaptation and extension of the *Read, Write, State-Reset* (RWS) model presented in [31]. The original RWS model introduces just three types of events that need to be captured in order to track provenance: *read events* (a data token was read by an actor), *write events* (a data token was written by an actor), and *state-reset* events (an actor's current state was reset, meaning that there are no dependencies between data tokens read by the actor before the reset and produced by the actor after the reset). This model enables capturing data dependencies correctly even if actors are stateful, i.e. data produced in a given firing depends on a sequence of data consumed in multiple previous firings. Examples of such stateful computations include daily average temperature computation or processing over a sliding window.

This model adapted to HyperFlow is even simpler than the original one because HyperFlow processes do not use the concept of ports.

A provenance event is a 5-tuple:

(*appId, processId, firingId, signalInstanceId, eventType*)

where:

1. *appId* identifies the workflow instance;
2. *processId* identifies the specific process that produced the event;
3. *firingId* identifies the particular firing of the process.
4. *signalInstanceId* is a tuple (σ, ι) (signal identifier and signal instance number) identifying the signal instance related to the event (for read or write events);
5. *eventType* is one of *read*, *write*, *state-reset*.

This model is insufficient in some important workflow scenarios, such as the thread-merge pattern shown in Fig. 9(a). This pattern is used to wait for a number of firings to finish, collect data elements produced by these firings (the *elem* signal), and emit a list of these elements once they have all been collected. Another signal (*cnt*) is used to control the processing loop.

We propose to extend the RWS model with an additional event types: *state-remove* The extended model – called '*Read–Write–State change*' – is summarized in Table 1. While *state-reset* removes all signal instances currently in the state of a process, the new *state-remove* event permits selective signal removals, resulting in a much more flexible model.

Fig. 9 presents two provenance traces for the scenario from Fig. 9(a): according to the original RWS model (9(b)) and the extended one (9(c)). In order to compare the expressiveness of the two models, let us formally define the provenance of a signal.

Let event(P, F, S, T) be a relation which denotes the occurrence of an event in a provenance trace, where P is the *appId* together with *processId*, F is the *firingId*, S denotes the *signalInstanceId*, and T is the *eventType*; for example: event$((app_1, p_1), 1, (cnt, 1), read)$. We define two other relations:

$$\begin{aligned}\text{in-state}(s, p, f_0) \equiv\ & \text{event}(p, f_1, s, read) \wedge \\ & \neg\, \text{event}(p, f_2, s, state - remove) \wedge \\ & \neg\, \text{event}(p, f_3, s, state - reset) \wedge \\ & f_1 \leq f_2, f_3 \leq f_0 \\ s_1 \text{ depends-on } s_2 \equiv\ & \text{event}(p_0, f_1, s_1, write) \wedge \\ & \text{in-state}(s_2, p_0, f_1).\end{aligned}$$

Relation in-state denotes that signal s belongs to state of process p in firing f_0, while relation depends-on means that signal s_2 was an input to produce signal s_1.

Let $\mathbb{S}$ be a set of all signal instances in a given provenance trace. Then the *provenance* of a signal $s_0 \in \mathbb{S}$ is defined as a directed acyclic graph:

$$\begin{aligned}&\text{provenance}(s_0) = (S, E) \quad \text{where:} \\ &\quad S \subseteq \mathbb{S} \quad \text{is a set such that:} \\ &\qquad s_0 \in S \\ &\qquad \text{if} \quad s \in S \wedge s \text{ depends-on } s_1 \quad \text{then} \quad s_1 \in S \\ &\quad E = \{(s_1, s_2) \colon s_1, s_2 \in S \wedge s_2 \text{ depends-on } s_1\}\end{aligned}$$

We also define the *origin* of s_0 as the set of original signals that s_0 was derived from:

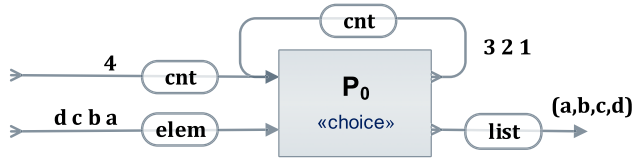
$$\text{origin}(s_0) = \{s \colon s \in \text{provenance}(s_0) = (S, E) \wedge s \in S \wedge (\_, s) \notin E\}.$$

Based on these definitions and the provenance trace from Fig. 9(c), the provenance and origin of signal *list*:1 are, respectively, as follows:

$$\begin{aligned}&\text{provenance}(list{:}1) = (\{elem{:}1, elem{:}2, elem{:}3, elem{:}4, list{:}1\}, \\ &\quad \{(elem{:}1, list{:}1), (elem{:}2, list{:}1), (elem{:}3, list{:}1), (elem{:}4, list{:}1)\}) \\ &origin(list{:}1) = \{elem{:}1, elem{:}2, elem{:}3, elem{:}4\}\end{aligned}$$

The same results according to the original RWS model would be as follows:

$$\begin{aligned}&\text{provenance}(list{:}1) = (\{cnt{:}1, cnt{:}2, cnt{:}3, cnt{:}4, elem{:}1, \\ &\quad elem{:}2, elem{:}3, elem{:}4, list{:}1\}, \\ &\quad \{(elem{:}1, list{:}1), (elem{:}2, list{:}1), (elem{:}3, list{:}1), (elem{:}4, list{:}1), \\ &\quad (cnt{:}1, list{:}1), (cnt{:}2, list{:}1), (cnt{:}3, list{:}1), (cnt{:}4, list{:}1)\}) \\ &origin(list{:}1) = \{cnt{:}1, cnt{:}2, cnt{:}3, cnt{:}4, elem{:}1, elem{:}2, elem{:}3, elem{:}4\}.\end{aligned}$$



(a) Thread merge pattern.

Proc. id	Firing id	Signal instance (name,id,value)	Event type
P_0	1	(cnt,1,4)	read
P_0	1	(elem,1,a)	read
P_0	1	(cnt,2,3)	write
P_0	2	(cnt,2,3)	read
P_0	2	(elem,2,b)	read
P_0	2	(cnt,3,2)	write
P_0	3	(cnt,3,2)	read
P_0	3	(elem,3,c)	read
P_0	3	(cnt,4,1)	write
P_0	4	(cnt,4,1)	read
P_0	4	(elem,4,d)	read
P_0	4	(list,1,[a,b,c,d])	write
P_0	5	—	s-reset

(b) Provenance trace—original RWS model adapted to HyperFlow.

Proc. id	Firing id	Signal instance (name,id,value)	Event type
P_0	1	(cnt,1,4)	read
P_0	1	(elem,1,a)	read
P_0	1	(cnt,2,3)	write
P_0	2	(cnt,1,4)	s-remove
P_0	2	(cnt,2,3)	read
P_0	2	(elem,2,b)	read
P_0	2	(cnt,3,2)	write
P_0	3	(cnt,2,3)	s-remove
P_0	3	(cnt,3,2)	read
P_0	3	(elem,3,c)	read
P_0	3	(cnt,4,1)	write
P_0	4	(cnt,3,2)	s-remove
P_0	4	(cnt,4,1)	read
P_0	4	(elem,4,d)	read
P_0	4	(cnt,4,1)	s-remove
P_0	4	(list,1,[a,b,c,d])	write
P_0	5	—	s-reset

(c) Provenance trace—extended RWS model.

Fig. 9. Comparison of provenance traces for the thread merge workflow pattern.

Intuitively, the first result is correct because the result `list:1` does not depend on the `cnt` signals which only play auxiliary role controlling the processing loop.

5. Workflow execution

5.1. HyperFlow enactment engine

The HyperFlow engine is responsible for the enactment of the process network describing a workflow, i.e. queuing signals, firing processes, and invoking the activity functions. It is important to note that other crucial aspects of the workflow execution – such as resource provisioning, deployment of workflow components, job scheduling, or data transfer – are delegated to services of the infrastructure. For example, an input signal may represent a file and contain an FTP link to the file's location. In such a case this information should be passed to an external service that will transfer the file to the appropriate computing node. We provide tools that assist in the workflow deployment and support workflow scheduling, data staging, and execution of workflow components in a cloud. Sections 5.2 and 5.3 briefly discuss these.

The HyperFlow engine is implemented as a lightweight `Node.js` application[6] (with the core code base of only about 200 kB, excluding workflow examples and external libraries). It requires the `Node` runtime and `Redis` [32] which is used to persist the state of running workflow instances.

The engine can be used as a command line tool called `hflow`. For example, a workflow can be executed using command `hflow run <wfdir>` where directory `wfdir` should contain the workflow implementation in files named `workflow.json` (workflow graph) and `functions.js` (workflow activities). The HyperFlow engine can also be used via a REST API, using any HTTP client. The command `hflow start-server` starts a new instance of the HyperFlow engine in a server mode and prints out the URI of so-called *app factory* which can be used to create new workflow instances using an HTTP POST request.

5.2. Workflow deployment

A workflow application can be orchestrated by the HyperFlow engine running on a user's laptop. In this case, the user is responsible for deployment of workflow application components, implementing their invocation from workflow activities, implementing data transfers, etc. However, HyperFlow also supports a deployment model that facilitates provisioning of computing resources from a cloud infrastructure and workflow execution in such an infrastructure. In this model, shown in Fig. 10, the entire HyperFlow runtime environment is deployed in the cloud alongside workflow application components. Workflow activities can be mapped to programs installed on Virtual Machines.

[6] The HyperFlow engine is available as open source at https://github.com/dice-cyfronet/hyperflow.

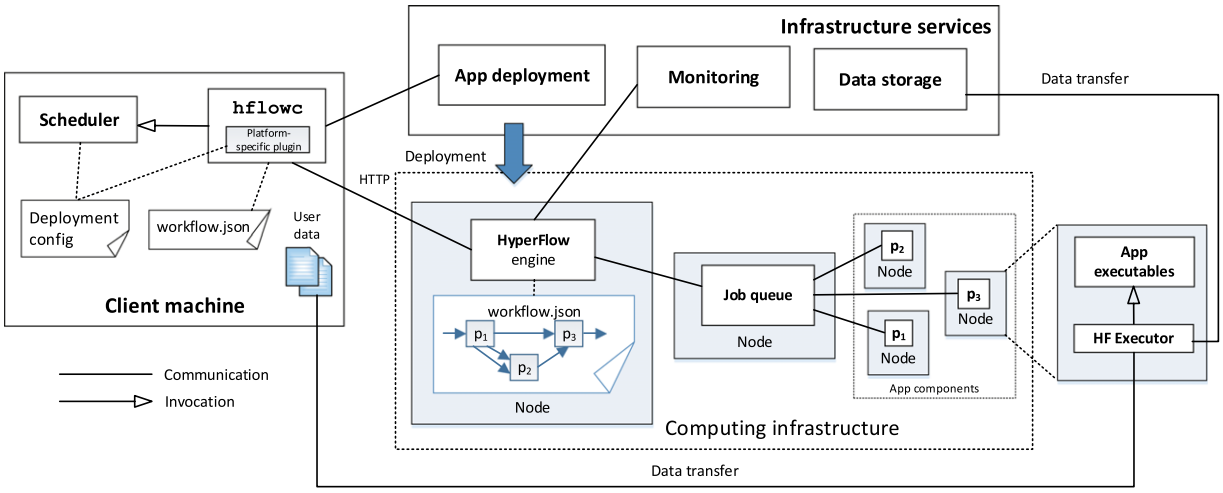


Fig. 10. Deployment of a HyperFlow workflow in a computing infrastructure.

In order to leverage this deployment option, the user needs to do the following steps: (1) prepare Virtual Machine images with workflow application executables. (2) Implement the workflow; if all activities are mapped to executables, the user only needs to implement the `workflow.json` file. (3) Configure the workflow and deployment details; the user needs to specify the mapping of workflow activities onto VM images as well as additional configuration specific to a particular cloud platform (support for different cloud platforms can be implemented as a plugin to HyperFlow).

The user runs a workflow prepared in this way using a special HyperFlow client `hflowc.` This client deploys the HyperFlow runtime environment and workflow application components in the target cloud. The HyperFlow engine orchestrates the workflow through a job queue. Workflow activities send job descriptions to the queue and wait for the notification of their completion. Executors deployed in the virtual machines fetch job descriptions, then, if needed, transfer files from a data store (dependent on a particular cloud environment), and invoke appropriate application binaries.

More details about this deployment solution, discussion of its advantages, as well as its application in two different cloud platforms have been published in [33].

5.3. Workflow scheduling

Simple workflow models, such as DAG, are conducive to easy scheduling algorithms. In HyperFlow, the workflow model is more complex. In particular, processes can be fired multiple times, unlike nodes in a DAG. Let us briefly discuss the consequences of this for scheduling of HyperFlow workflows. In general, there are two cases: (1) it is known prior to workflow execution how many times each workflow activity will fire. (2) such knowledge is not available. In the first case, we can build a DAG out of a HyperFlow workflow and pass it to a scheduler. Such techniques have been applied to workflows with more complex constructs, e.g. for unrolling loops to a DAG [34]. In the second case, we can rely on dynamic/online scheduling strategies such as work-sharing or work-stealing. In such a case firings of workflow activities add new jobs to a pool which can be dynamically balanced among available resources. However, even in such cases historical information from previous executions can be used to deal with dynamic workflow behavior [34]. In depth research on such scheduling techniques in HyperFlow is a work in progress. So far we have published two papers focusing on HyperFlow-based applications that also discuss some scheduling aspects. In [35], a workflow is analyzed as a DAG of tasks, and we apply graph clustering in order to improve the computation/communication ratio. In [36], a workflow is treated as a bag-of-tasks application. We introduce a performance model for such applications which allows the scheduler to compute the number of instances required to meet a deadline.

5.4. Distributed execution

The REST API is crucial to enable a unique feature of the HyperFlow engine which is the capability to **link multiple workflow instances**, possibly controlled by separate HyperFlow engines, so that they can exchange signals. Effectively, workflows can be executed in a completely distributed and decentralized manner, without the control of a central enactment engine. A connection between different workflow instances can be made using the mechanism of *remote sinks* which can be associated with any signal. A remote sink is simply an URI of a (remote) workflow instance. Whenever a signal is produced by a process, in addition to being emitted to its local sinks, it is also sent via the REST API to its remote sinks.

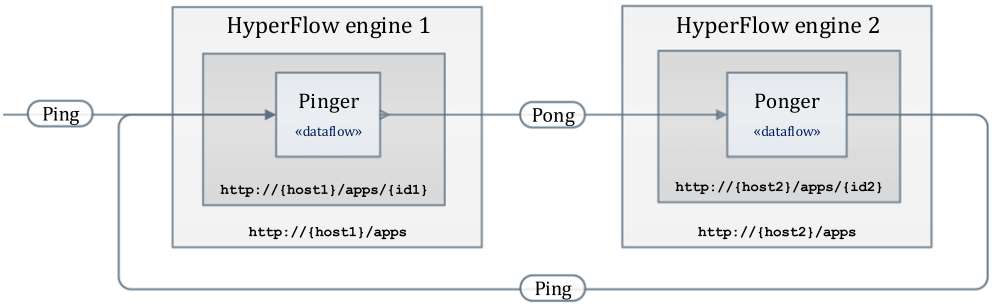
Let us study a simple example – the Distributed Ping-Pong workflow – in order to illustrate the distributed enactment of a workflow and the utilization of the REST API. Fig. 11 shows an example deployment of two trivial workflows in two instances of the HyperFlow engine. The workflows – Pinger and Ponger – exchange signals triggering each other in an infinite loop. The figure also shows the URIs of the involved REST resources, and the sequence of HTTP requests used to create the Pinger and Ponger, connect them together by updating the remote sinks of their respective signals, and sending the initial signal to start the workflow.

5.5. Provenance tracking

The HyperFlow engine automatically logs provenance events compliant with the Read–Write–State-Reset (RWS) provenance model [31]. Such a log can be imported to a structured provenance model in order to support complex provenance queries. The details of the provenance model are described in Section 4.2.

6. Workflow examples

This section presents several example workflows implemented in HyperFlow. Existing workflows previously implemented in other workflow systems – Taverna, Kepler and Pegasus – were chosen and re-implemented in HyperFlow, in order to compare



URI short	URI full	Meaning
{appfactory1}	http://{host1}/apps	*Engine 1 app factory URI*
{appfactory2}	http://{host2}/apps	*Engine 2 app factory URI*
{app1}	http://{host1}/apps/{id1}	*Pinger instance URI*
{app2}	http://{host2}/apps/{id2}	*Ponger instance URI*

(a) Distributed Ping-Pong workflow illustration and URIs.

```
{
  "name": "Wf_RemotePing",
  "processes": [ {
    "name": "Pinger",
    "type": "dataflow",
    "function": "inc",
    "ins":  [ "Ping" ],
    "outs": [ "Pong" ]
  } ],
  "signals": [ {
    "name": "Ping",
  }, {
    "name": "Pong"
  } ],
  "ins":  [ "Ping" ],
  "outs": [ "Pong" ]
}
```

(b) Pinger description (`Pinger.json`).

```
{
  "name": "Wf_RemotePong",
  "processes": [ {
    "name": "Ponger",
    "type": "dataflow",
    "function": "inc",
    "ins":  [ "Pong" ],
    "outs": [ "Ping" ]
  } ],
  "signals": [ {
    "name": "Ping",
  }, {
    "name": "Pong"
  } ],
  "ins":  [ "Pong" ],
  "outs": [ "Ping" ]
}
```

(c) Ponger description (`Ponger.json`).

Operation & URI	Body	Effect
POST {appfactory1}	Pinger.json	Create Pinger instance, return URI {app1}
POST {appfactory2}	Ponger.json	Create Ponger instance, return URI {app2}
PUT {app1}/sigs/Ping/remotesinks	{"uri":{app2}}	Connect signal Ping from app1 to app2
PUT {app2}/sigs/Pong/remotesinks	{"uri":{app1}}	Connect signal Pong from app2 to app1
POST {app1}	Ping signal data	Send the initial signal to start Ping-Pong

(d) REST protocol steps for creating and running the workflow.

Fig. 11. Distributed Ping-Pong workflow deployed in and run by two instances of the HyperFlow engine.

the different approaches. The workflows are available in the HyperFlow distribution.[7]

6.1. Biological pathways workflow: eliminating shim nodes

The first example workflow, originally implemented in Taverna 2,[8] computes biological pathways involving genes passed as workflow inputs using their Unigene identifiers. The HyperFlow implementation, shown in Fig. 12, consists of five processes, each invoking the KEGG REST API[9] in order to convert the gene identifiers (`ConvertId`), retrieve the identifiers of pathways involving a gene (`GetPathWayByGene`), retrieve pathway information (`GetPathWayEntry`) along with its graphical representation (`GetPathWaImage`), and retrieve the gene description (`GetGeneDescription`). The activity of each process is implemented as a JavaScript function which contains about 15–20 lines of code comprising the invocation of the appropriate REST service and required transformations (glue code).

Taverna's implementation, on the other hand, has as many as 13 processing nodes, five of which invoke the actual domain components (REST services), while the remaining 8 are shim nodes implemented as *Beanshell services*. This is a significant number given that the required transformations are rather trivial (simple string manipulations and splitting collections into individual items). More nodes in the workflow add programming effort related to additional lines of code (e.g. for reading from input ports

[7] https://github.com/dice-cyfronet/hyperflow/tree/develop/examples.

[8] The Taverna implementation is available through *myexperiment.org*: http://www.myexperiment.org/workflows/2673.

[9] KEGG API, http://www.kegg.jp/kegg/rest.

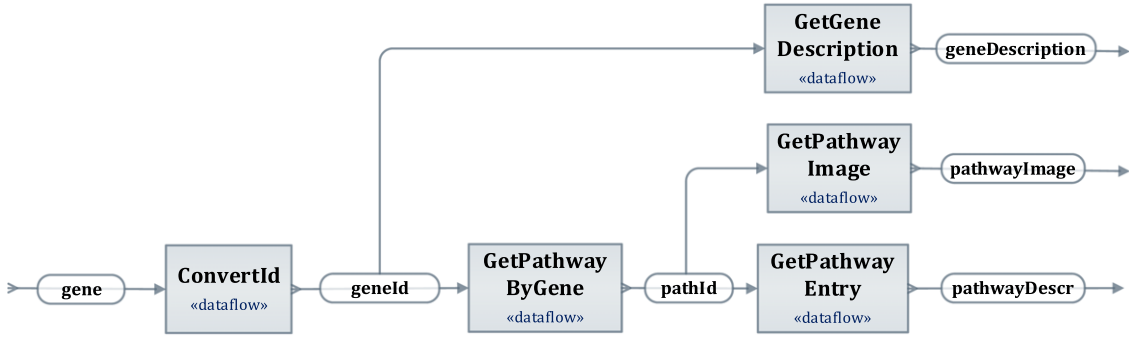


Fig. 12. The biological pathways workflow: HyperFlow implementation.

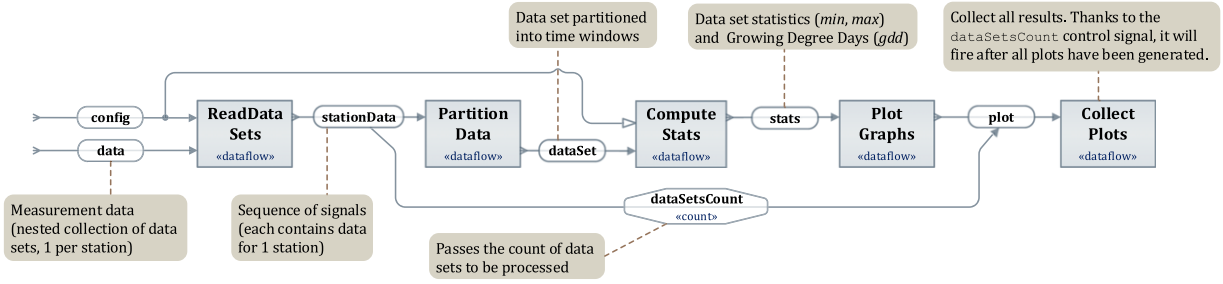


Fig. 13. The Comet workflow: HyperFlow implementation.

and writing to output ports), filling out forms to configure the beanshell services, etc.

In the HyperFlow implementation all glue code is easily hidden in the implementation of workflow activities, hence five processes are sufficient, and the code itself is much more concise. Let us note that we could have chosen to exactly replicate the structure of the Taverna workflow and implement glue code in separate HyperFlow processes. The low-level programming approach gives us this flexibility, allowing to choose the granularity of workflow nodes and the trade-offs related with it. In Taverna and similar workflow systems this is not possible because most operations are supported by dedicated components (Taverna's *REST service* used in the pathways workflow being an example) which allow only limited customization through pre-defined configuration options. Consequently, glue code can only be added as separate shim nodes.

Finally, let us consider the question whether hiding workflow-specific glue-code in the implementation of workflow activities might reduce the reusability of the involved workflow components? Not only is this not the case but on the contrary: this approach facilitates reusability because the reusable code can easily be factored out as a separate JavaScript function which can be invoked in various workflows from the respective workflow activities that only add a small amount of workflow-specific glue code.

6.2. The comet workflow: parallel processing of nested data collections

The next example is the Comet workflow, originally implemented in the Kepler workflow system [37]. This workflow takes collections of measurement data, divides them into subsets according to the location, and for each subset, it calculates statistics and generates charts. The HyperFlow implementation consists of five processes, as illustrated in Fig. 13.

The input to the workflow is an XML file with measurement data grouped according to stations and locations (US states and counties). The first process (`ReadDataSets`) reads the file and divides the data into sets, one for each station. These sets are emitted as a collection of signals. Each data set is then processed by the pipeline of three processes: `PartitionData` which divides a set into 24-h time windows; `ComputeStats` which calculates data statistics (min, max and growing degree-day), separately for each time window; and, finally, `PlotGraphs` which invokes the R tool and generates a set of charts visualizing the statistics for each station. The `CollectPlots` process is added for cleanup purposes. Thanks to the `dataSetsCount` signal, it is only fired after all data sets have been processed.

The workflow involves significant amount of XML transformations which in the original Kepler implementation (based on the standard Process Networks Director) were handled by shim nodes, so that the workflow had as many as 15 activities. In an effort to simplify this and reduce the number of shim nodes, a new Kepler director, called *Comad*, specifically dedicated to data collections, has been implemented [37]. The Comet workflow implemented with the Comad director has only five activities which is the same as the HyperFlow version. However, the HyperFlow implementation is significantly simpler: it only requires 60 lines of the JSON description and 150 lines of the JavaScript implementation code for workflow activities. By contrast, implementing a new Kepler Director alone is a significant effort, and its orientation specifically towards data collections causes some limitations such as the requirement of a unified data format for representing collections. The Comad director also does not support data parallelism, unlike HyperFlow whose implementation supports both pipelining and data parallelism. 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. Fig. 14 illustrates this, showing the total workflow execution time depending on the value of `parlevel` for process `PlotGraphs.`

The implementation of new Kepler Actors also involves Kepler-specific boilerplate code and configuration files. The implementation of HyperFlow workflow activities, on the other hand, utilizes only four standard and widely known libraries (such as xpath and DOM parser), familiar to many programmers. In fact, once the workflow has been implemented, we have decided to refactor its code into a stand-alone `Node.js` application. The

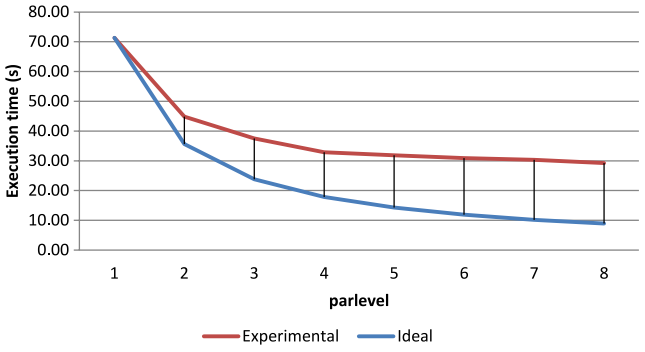


Fig. 14. The Comet workflow execution time depending on the value of the `parlevel` property of the `PlotGraphs` process. (Intel i7 2.30 GHz/8 cores.)

goal of this simple experiment was to investigate how much refactoring effort would be required and how much HyperFlow-specific code – representing the additional development effort imposed by HyperFlow – would be removed. As it turned out, the code required almost no refactoring. Basically only a main control loop has been added to the `functions.js` file and it immediately could be invoked as a regular `Node.js` application producing the same results (however, the exact semantics of HyperFlow workflow processing was not replicated using `Node.js` control constructs—it would require much more effort). The only change which felt desirable – albeit was not indispensable – was the refactoring of function signatures in order to replace the `ins` and `outs` arrays with regular function parameters. In other words, the code written for the workflow was immediately reusable in other applications, outside the HyperFlow context. From the reverse perspective, this experiment proves that the implementation of the workflow is very similar to 'regular' programming, adding very little overhead due to HyperFlow-specific code.

6.3. Montage: executing large-scale workflow on the cloud

The final example is the widely known and studied Montage workflow [38], originally implemented in the Pegasus system [16]. Montage produces high-quality images of celestial objects by transforming and merging thousands of smaller images. The original Pegasus implementation represents the workflow as a Directed Acyclic Graph (DAG) using an XML-based format (DAX). All workflow activities are Montage executables.[10]

In order to exactly replicate the Pegasus semantics of the workflow, we used the original DAG generator from Pegasus, and transformed the DAX format into the JSON representation of the HyperFlow model.[11] Listing 1 presents a fragment of the workflow description for a small Montage workflow comprising nearly 10,000 processes representing program executions and over 23,000 signals denoting input, output and intermediate files.

Listing 1: Montage scientific workflow: the HyperFlow implementation (fragment).

```
{
  "name": "Montage_10k",
  "functions": [ {
    "name": "amqp_command", // sends a task to the
        cloud via a message queue
    "module": "functions"
  } ],
  "processes": [ {
    "name": "mProjectPP",
    "function": "amqp_command",
    "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_14365
           3_22436.fits -1.5s 60s gaussian -out
           shrunken_20090720_143653_22436.jpg"
    },
    "ins": [ 22960 ],
    "outs": [ 22961 ]
  } ],
  "signals": [ { // here signals represent files
      and contain only file names
     "name": "2mass-atlas-980529s-j0150174.fits", }
          // ... about 23k more signals
  } ],
  "ins": [ ... ],
  "outs": [ ... ]
}
```

The Montage workflow has been executed on virtual machines deployed in the cloud using the deployment solution described in Section 5.2. The same workflow description has been successfully applied to run the Montage workflow on a local computer, in a private cloud with a shared file system used as a data store, and on an Amazon EC2 cloud using S3 for storing files. This shows an important feature of the HyperFlow programming approach: workflow activity functions provide a programmatic abstraction which makes the workflow description independent of the underlying runtime environment. In Pegasus and several other workflow systems the workflow description is essentially *abstract*. Only before execution it is adapted to a particular runtime environment, specific instances of services, etc. The adaptation implies rewriting the workflow graph and typically adding more auxiliary activities, e.g. for data transfer.

In HyperFlow, the workflow description is **always executable**, because the way the actual processing is implemented is hidden in functions implementing workflow activities. In the extreme case a function can be only a mock emulating the actual execution. Such an approach greatly benefits the workflow engineering process, in particular its testability. In the Montage case, the specifics of a particular runtime environment were hidden in the function's implementation, the workflow configuration (passed to the function) and in the implementation of VM-side executors.

7. Conclusion

The variety of existing workflow programming approaches and workflow systems is justified by diverse user groups and needs, as well as the multitude of science domains: while workflow systems are designed to be generic, in practice they tend to focus on one or two disciplines in terms of addressing a particular user community and supporting implementations of domain-specific workflow components.

The HyperFlow approach also aims at supporting a particular group of cases where workflow developers are skilled programmers desiring a productive workflow programming ecosystem. To this end, HyperFlow provides a workflow programming approach that combines a declarative description of the workflow graph

[10] More information about the executables can be found at http://montage.ipac.caltech.edu/docs/gridtools.html.

[11] The transformation software is available in the HyperFlow distribution: http://github.com/dice-cyfronet/hyperflow.

with low-level programming of workflow activities in a mainstream programming language and runtime platform. The HyperFlow model of computation enables the developer to implement complex workflow patterns using a simple and concise syntax. While the choice of a particular programming language for implementing workflow activities may seem to be a restriction imposed on the developer, this is not the case. The activity function merely provides a convenient framework for the actual implementation of the underlying processing: via a call to a web service, a submission of a job to a job scheduler, or an invocation of an external program written in one's favorite programming language (thereby completely bypassing the JavaScript code). The developer has a choice whether the glue code should be hidden in domain-specific activities, or exposed as separate shim nodes. The produced code is also more reusable because it depends only on the `Node.js` runtime environment, not on the HyperFlow itself.

As far as the HyperFlow runtime platform is concerned, the low-level programming capability again makes it easy to interface any execution environment and delegate the actual processing to external computing resources using APIs which are probably already implemented in libraries given the large community involved in the chosen runtime platform—`Node.js`. For example, `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 services (e.g. in Azure and Heroku). `Node.js` is also typically one of the first runtime environments supported by newly emerging cloud offerings, IBM Bluemix being an example.

So far HyperFlow has become a part of several larger systems where it has been used for a number of applications. The HyperFlow engine is the basis of a service for execution of scientific workflows in the PL-Grid infrastructure [39].[12] Currently supported applications include a workflow-based solver for finite-element meshes [35] which can be applied to diverse problems. In the EU FP7 funded project PaaSage,[13] HyperFlow is integrated with the PaaSage Platform middleware in order to enable multi-cloud execution and autoscaling of large-scale scientific applications. In the ISMOP project,[14] funded by the Polish National Centre for Research and Development, HyperFlow is used to enact workflows in a flood decision support system [36].

Acknowledgments

This work was partially supported by the National Centre for Research and Development (NCBiR), Poland, project PBS1/B9/18/2013; by the EU-FP7 project PaaSage; and by AGH University of Science and Technology, Faculty of Computer Science, Electronics and Telecommunications, Project No. 11.11.230.124.

References

[1] E. Deelman, D. Gannon, M. Shields, I. Taylor, Workflows and e-science: An overview of workflow system features and capabilities, Future Gener. Comput. Syst. 25 (5) (2009) 528–540.
[2] A. Belloum, M. Inda, D. Vasunin, V. Korkhov, Z. Zhao, H. Rauwerda, T.M. Breit, M. Bubak, L.O. Hertzberger, et al., Collaborative e-science experiments and scientific workflows, IEEE Internet Comput. 15 (4) (2011) 39–47.
[3] J. Goecks, A. Nekrutenko, J. Taylor, et al., Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences, Genome Biol. 11 (8) (2010) R86.
[4] E. Deelman, Y. Gil, Managing large-scale scientific workflows in distributed environments: Experiences and challenges, in: e-Science, 2006, p. 144.
[5] S. Tilkov, S. Vinoski, Node. js: Using JavaScript to build high-performance network programs, IEEE Internet Comput. 14 (6) (2010).
[6] B. Balis, Increasing scientific workflow programming productivity with hyperflow, in: Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science, WORKS'14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 59–69. http://dx.doi.org/10.1109/WORKS.2014.10.
URL: http://dx.doi.org/10.1109/WORKS.2014.10.
[7] J. Qin, T. Fahringer, Scientific Workflows: Programming, Optimization, and Synthesis with ASKALON and AWDL, Springer, 2012.
[8] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurr. Comput.: Pract. Exper. 18 (10) (2006) 1039–1065.
[9] T. Oinn, P. Li, D.B. Kell, C. Goble, A. Goderis, M. Greenwood, D. Hull, R. Stevens, D. Turi, J. Zhao, Taverna/myGrid: aligning a workflow system with the life sciences community, in: I. Taylor, E. Deelman, D. Gannon, M. Shields (Eds.), Workflows for e-Science, Springer, New York, Secaucus, NJ, USA, 2007, pp. 300–319.
[10] I. Taylor, M. Shields, I. Wang, A. Harrison, The triana workflow environment: Architecture and applications, in: Workflows for e-Science, Springer, New York, Secaucus, NJ, USA, 2007, pp. 320–339.
[11] M. Hardt, T. Jejkal, I. Campos, E. Fernandez, A. Jackson, D. Nielsson, B. Palak, M. Płociennik, Transparent access to scientific and commercial clouds from the KEPLER workflow engine, Comput. Inform. 31 (1) (2012) 119–134.
[12] K.V. Knyazkov, S.V. Kovalchuk, T.N. Tchurov, S.V. Maryin, A.V. Boukhanovsky, CLAVIRE: e-Science infrastructure for data-driven computing, J. Comput. Sci. 3 (6) (2012) 504–510.
[13] R. Cushing, S. Koulouzis, A. Belloum, M. Bubak, Applying workflow as a service paradigm to application farming, Concurr. Comput.: Pract. Exper. 26 (6) (2014) 1297–1312.
[14] P. Missier, S.S. Sahoo, J. Zhao, C. Goble, A. Sheth, Janus: From workflows to semantic provenance and linked open data, in: Provenance and Annotation of Data and Processes, Springer, 2010, pp. 129–141.
[15] P. Missier, K. Belhajjame, J. Zhao, M. Roos, C. Goble, Data lineage model for Taverna workflows with lightweight annotation requirements, in: Provenance and Annotation of Data and Processes, Springer, 2008, pp. 17–30.
[16] E. Deelman, G. Mehta, G. Singh, M.-H. Su, K. Vahi, Pegasus: Mapping large-scale workflows to distributed resources, in: Workflows for e-Science, Springer, New York, 2007, pp. 376–394.
[17] P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai, I. Marton, WS-PGRADE/gUSE generic DCI gateway framework for a large variety of user communities, J. Grid Comput. 10 (4) (2012) 601–630.
[18] A. Balasko, Z. Farkas, P. Kacsuk, Building science gateways by utilizing the generic WS-PGRADE/gUSE workflow system, Comput. Sci. J. 14 (2) (2013) 307–325.
[19] G. Terstyanszky, T. Kukla, T. Kiss, P. Kacsuk, A. Balasko, Z. Farkas, Enabling scientific workflow sharing through coarse-grained interoperability, Future Gener. Comput. Syst. 37 (2014) 46–59.
[20] K. Plankensteiner, R. Prodan, M. Janetschek, T. Fahringer, J. Montagnat, D. Rogers, I. Harvey, I. Taylor, Á Balaskó, P. Kacsuk, Fine-grain interoperability of scientific workflows in distributed computing infrastructures, J. Grid Comput. 11 (3) (2013) 429–455.
[21] I. Foster, M. Hategan, J.M. Wozniak, M. Wilde, B. Clifford, Swift: A language for distributed parallel scripting, Parallel Comput. 37 (9) (2011) 633–652.
[22] K. Maheshwari, J. Montagnat, Scientific workflow development using both visual and script-based representation, in: Services 2010, IEEE 6th World Congress on Services, IEEE, 2010, pp. 328–335.
[23] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, M.B. Fornarino, A data-driven workflow language for grids based on array programming principles, in: WORKS09, Proc. 4th Workshop on Workflows in Support of Large-Scale Science, ACM, 2009, p. 7.
[24] M. Baranowski, A. Belloum, M. Bubak, M. Malawski, Constructing workflows from script applications, Sci. Program. 20 (4) (2012) 359–377.
[25] M. Malawski, T. Bartynski, M. Bubak, A tool for building collaborative applications by invocation of grid operations, in: M. Bubak, G.D. van Albada, J. Dongarra, P.M.A. Sloot (Eds.), Computational Science—ICCS 2008, 8th International Conference, Kraków, Poland, June 23–25, 2008, Proceedings, Part III, in: Lecture Notes in Computer Science, vol. 5103, Springer, 2008, pp. 243–252.
[26] Y. Li, M. Mascagni, Analysis of large-scale grid-based Monte Carlo applications, Int. J. High Perform. Comput. Appl. 17 (4) (2003) 369–382.
[27] C. Pautasso, G. Alonso, Parallel computing patterns for grid workflows, in: WORKS'06: Workshop on Workflows in Support of Large-Scale Science, IEEE, 2006, pp. 1–10.
[28] G. Kahn, The semantics of a simple language for parallel programming, in: Information Processing, North-Holland, 1974, pp. 471–475.
[29] E.A. Lee, T.M. Parks, Dataflow process networks, Proc. IEEE 83 (5) (1995) 773–801.
[30] R. Bocchino, V. Adve, S. Adve, M. Snir, Parallel programming must be deterministic by default, in: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, USENIX Association, 2009, 4–4. Available online: https://www.usenix.org/legacy/events/hotpar09/tech/.

[12] This work is done within the PL-Grid Core project, http://www.plgrid.pl.
[13] http://www.paasage.eu.
[14] http://www.ismop.edu.pl.

[31] S. Bowers, T.M. McPhillips, B. Ludäscher, S. Cohen, S.B. Davidson, A model for user-oriented data provenance in pipelined scientific workflows, in: L. Moreau, I.T. Foster (Eds.), Provenance and Annotation of Data, International Provenance and Annotation Workshop, IPAW 2006, in: Lecture Notes in Computer Science, vol. 4145, Springer, Chicago, IL, USA, 2006, pp. 133–147.
[32] J.L. Carlson, Redis in Action, Manning Publications Co., 2013.
[33] B. Balis, K. Figiela, M. Malawski, M. Pawlik, M. Bubak, A lightweight approach for deployment of scientific workflows in cloud infrastructures, in: Parallel Processing and Applied Mathematics, 11th International Conference, PPAM 2015, Revised Selected Papers, in: Lecture Notes in Computer Science, Springer, 2015, in press.
[34] R. Prodan, M. Wieczorek, Negotiation-based scheduling of scientific grid workflows through advance reservations, J. Grid Comput. 8 (4) (2010) 493–510.
[35] B. Balis, K. Figiela, M. Malawski, K. Jopek, Leveraging workflows and clouds for a multi-frontal solver for finite-element meshes, Procedia Comput. Sci. 51 (2015) 944–953.
[36] B. Balis, M. Kasztelnik, M. Malawski, P. Nowakowski, B. Wilk, M. Pawlik, M. Bubak, Execution management and efficient resource provisioning for flood decision support, Procedia Comput. Sci. 51 (2015) 2377–2386.
[37] L. Dou, D. Zinn, T. McPhillips, S. Kohler, S. Riddle, S. Bowers, B. Ludascher, Scientific workflow design 2.0: Demonstrating streaming data collections in Kepler, in: IEEE 2011 Data Engineering (ICDE) Conference, IEEE, 2011, pp. 1296–1299.
[38] G.B. Berriman, E. Deelman, et al., Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand, in: Astronomical Telescopes and Instrumentation, International Society for Optics and Photonics, 2004, pp. 221–232.
[39] M. Bubak, J. Kitowski, K. Wiatr, EScience on Distributed Computing Infrastructure: Achievements of PLGrid Plus Domain-specific Services and Tools, in: Lecture Notes in Computer Science, vol. 8500, Springer, 2014.

Bartosz Balis, Ph.D. in Computer Science, is an Assistant Professor at the Department of Computer Science AGH. He is co-author of around 70 international publications including journal articles, conference papers, and book chapters. His research interests include environments for eScience, scientific workflows, grid and cloud computing. He has been a participant of international research projects including EU-IST CrossGrid, CoreGRID, K-Wf Grid, ViroLab, Gredia, UrbanFlood and PaaSage. He has been a Member of Program Committee for conferences (e-Science 2006, ICCS 2007–2015, ITU Kaleidoscope 2013–2015).