Introducing SWIRL: An Intermediate Representation Language for Scientific Workflows

Iacopo Colonnelli[1], Doriana Medić[1(✉)], Alberto Mulone[1], Viviana Bono[1], Luca Padovani[2], and Marco Aldinucci[1]

[1] University of Turin, Turin, Italy
{iacopo.colonnelli,doriana.medic,alberto.mulone,
viviana.bono,marco.aldinucci}@unito.it
[2] University of Camerino, Camerino, Italy
luca.padovani@unicam.it

Abstract. In the ever-evolving landscape of scientific computing, properly supporting the modularity and complexity of modern scientific applications requires new approaches to workflow execution, like seamless interoperability between different workflow systems, distributed-by-design workflow models, and automatic optimisation of data movements. In order to address this need, this article introduces SWIRL, an intermediate representation language for scientific workflows. In contrast with other product-agnostic workflow languages, SWIRL is not designed for human interaction but to serve as a low-level compilation target for distributed workflow execution plans. The main advantages of SWIRL semantics are low-level primitives based on the send/receive programming model and a formal framework ensuring the consistency of the semantics and the specification of translating workflow models represented by Directed Acyclic Graphs (DAGs) into SWIRL workflow descriptions. Additionally, SWIRL offers rewriting rules designed to optimise execution traces, accompanied by corresponding equivalence. An open-source SWIRL compiler toolchain has been developed using the ANTLR Python3 bindings.

Keywords: Hybrid workflow · Interoperability · Formal methods

1 Introduction

Workflows have been widely used to model large-scale scientific workloads. The explicit definition of true dependencies between subsequent steps allows inferring concurrent execution strategies automatically, improving performances, and transferring input and output data wherever needed, fostering large-scale distributed executions. However, current Workflow Management Systems (WMSs) struggle to keep up with the ever-more demanding requirements of modern scientific applications, such as *interoperability* between different systems, *distributed-by-design* workflow models, and *automatic optimisation of data movements*.

A. Platzer et al. (Eds.): FM 2024, LNCS 14933, pp. 226–244, 2025.
https://doi.org/10.1007/978-3-031-71162-6_12

With the advent of BigData, adopting a proper *data management* strategy has become a crucial aspect of large-scale workflow orchestration. Avoiding unnecessary data movements and coalescing data transfers are two established techniques for performance optimisation in distributed executions. Moving computation near data to remove the need for data transfers is the underlying principle of several modern approaches to large-scale executions, like Resilient Distributed Datasets [42] and in-situ workflows [4].

WMSs' *interoperability* is an open problem in scientific workflows, which hinders reusability and composability. Despite several attempts to model product-agnostic workflow languages [11] and representations [28] present in the literature, these solutions capture only a subset of features, forcing WMSs to reduce their expressive power in the name of portability. The main issue in unifying workflow representations resides in the heterogeneity of different WMSs' APIs and programming models tailored to the needs of a domain experts. Conversely, moving the interoperability efforts to the lower level of the workflow execution plan representation is a promising but still relatively unexplored alternative.

The *heterogeneity* in contemporary hardware resources and their features, further exacerbated by the end-to-end co-design approach [30], requires WMSs to support a large ecosystem of execution environments (from HPC to cloud, to the Edge), optimisation policies (performance vs. energy efficiency) and computational models (from classical to quantum). However, maintaining optimised executors for such diverse execution targets is an overarching effort. In this setting, a just-in-time compilation of target-specific execution bundles, optimised for a single workflow running in a single execution environment, would be a game-changing approach. Indeed, this approach allows for the efficient use of resources, as the compilation is done at the time of execution, taking into account the specific characteristics of the execution environment. It also ensures the effectiveness of the execution, as the compiled bundle is optimised for the specific workflow, leading to improved performance.

This work presents SWIRL, a "Scientific Workflow Intermediate Representation Language". Unlike other product-agnostic workflow languages, SWIRL is not intended for human interaction but serves as a low-level compilation target for distributed workflow execution plans. It models the execution plan of a location-aware workflow graph as a distributed system with send/receive communication primitives. This work provides a formal method to encode a workflow instance into a distributed execution plan using these primitives, promoting interoperability and composability of different workflow models. It also includes a set of rewriting rules for automatic optimisation of data communications with correctness and consistency guarantees. The optimised SWIRL representation can then be compiled into one or more self-contained executable bundles, making it adaptable to specific execution environments and embracing heterogeneity.

The SWIRL implementation follows the same line as the theoretical approach, separating scientific workflows' design and runtime phases. A SWIRL-based compiler translates a workflow system `W` to a high-performance, self-contained workflow execution bundle based on send/receive communication pro-

tocols and runtime libraries, which can easily be included in a Research Object [5], significantly improving reproducibility.

In detail, Sect. 2 introduces a generic formalism for representing distributed scientific workflow models, while the related work and the comparison with the SWIRL language is given in Sect. 2.1. Section 3 introduces the SWIRL semantics, and Sect. 4 derives the rewriting rules used for optimisation. Section 5 describes the implementation of the SWIRL compiler toolchain while Sect. 6 shows how to model the 1000 Genomes workflow [35], a Bioinformatics pipeline aiming at fetching, parsing and analysing data from the 1000 Genomes Project [39] into SWIRL system. Finally, Sect. 7 concludes the article. Full proofs and additional material can be found in [10] while the experiment is in [9].

2 Background and Related Work

This section gathers the related work (Sect. 2.1) and introduces a formal representation of scientific workflows (Sect. 2.2) and their mapping onto distributed and heterogeneous execution environments (Sect. 2.3).

2.1 Related Work

Location-Aware WMSs. Grid-native WMSs typically support distributed workflows out of the box, providing automatic scheduling and data transfer management across multiple execution locations. However, all the orchestration aspects are delegated to external, grid-specific technologies, limiting the spectrum of supported execution environments. For instance, Triana [36], Askalon [14] and Pegasus [12] delegate tasks offloading and data transfers to the GAP interface [37], the GLARE library [34], and HTCondor [38], respectively.

Recently, a new class of location-aware WMSs is bringing advantages in performance and costs of workflow executions on top of heterogeneous distributed environments. StreamFlow [8] allows users to explicitly map each step onto one or more locations in charge of its execution. It relies on a set of connectors to support several execution environments, from HPC queue managers to microservices orchestrators. Jupyter Workflow [7] transforms a sequential computational notebook into a distributed workflow by mapping each cell into one or more execution locations, semi-automatically extracting inter-cell data dependencies from the code, and delegating the runtime orchestration to StreamFlow. Mashup [32] automatically maps each workflow step onto the best-suited location, choosing between traditional Cloud VMs and serverless platforms.

Each tool has its own strategy to derive an execution plan from a workflow graph without relying on an explicit and consolidated intermediate representation. Moreover, none of them formalise this derivation process, hiding its details inside the WMS's codebase. Instead, relying on a common intermediate language like SWIRL would allow interoperability between different tools and formal correctness guarantees on the adopted optimisation strategies.

Formal Models for Distributed Workflows. In the literature, the number of different WMSs is notable [3], however, up to our knowledge, there are only a few WMS for which formal models have been developed: Taverna [41], employing the lambda calculus [25] to define the workflow language in functional terms; Kepler [21] adopting Process Networks [18] and BPEL [26], where the workflow language is formalised with Petri Nets. YAWL [1] is another workflow language based on Petri Nets extended with constructs to address the multiple instances, advanced synchronisation, and cancellation patterns. It provides a detailed representation of workflow patterns [2] supported by an open-source environment.

Process algebra, in particular, different versions of π-calculus [33] are suited to model the workflow system due to the ability of processes to change their structure dynamically. A class of workflow patterns has been precisely defined using the execution semantics of π-calculus, in [29], while the basic control flow constructs modelled by π-calculus are given in [13]. A distributed extension of π-calculus [15] is examined as a formalisation for distributed workflow systems in [23], providing a discussion on the flexibility of the proposed representation. Aside from π-calculus, CCS (Calculus of Communication Systems) [24] models Web Service Choreography Interface descriptions.

2.2 Scientific Workflow Models

A generic workflow can be represented as a *directed bipartite graph*, where the nodes refer to either the computational *steps* of a modular application or the *ports* through which they communicate, and the edges encode *dependency relations* between steps.

Definition 1. *A workflow is a directed bipartite graph* $W = (S, P, \mathcal{D})$ *where* S *is the set of steps,* P *is the set of ports, and* $\mathcal{D} \subseteq (S \times P) \cup (P \times S)$ *is the set of dependency links.*

In the considered graph, one port can have multiple output edges meaning that more steps are dependent on it. The sets of input/output ports (steps) of a step (port) are defined with the following definition.

Definition 2. *Given a workflow* $W = (S, P, \mathcal{D})$,*a step* $s \in S$ *and a port* $p \in P$*, the sets of input and output ports of* s *are denoted with* $In(s)$ *and* $Out(s)$*, respectively, and defined as:*

$$In(s) = \{p' \mid (p', s) \in \mathcal{D}\} \qquad Out(s) = \{p' \mid (s, p') \in \mathcal{D}\}$$

while the sets of input and output steps of p *are denoted with* $In(p)$ *and* $Out(p)$*, respectively, and defined as:*

$$In(p) = \{s' \mid (s', p) \in \mathcal{D}\} \qquad Out(p) = \{s' \mid (p, s') \in \mathcal{D}\}$$

Traditionally, scientific workflows are modelled using a dataflow approach, i.e., following *token-pushing* semantics in which tokens carry *data values*. The step executions are enabled by the presence of tokens in their input ports and

produce new tokens in their output ports. In general, a single workflow model can generate infinite *workflow instances*. Different instances preserve the same graph structure but differ in the values carried by each token.

Definition 3. *A workflow instance is a tuple* $(W, D, \mathcal{I})$ *where* $W = (S, P, \mathcal{D})$ *is a workflow,* D *is a set of data elements, and* $\mathcal{I} \subseteq (D \times P)$ *is a mapping relation connecting each data element* $d \in D$ *to the port* $p \in P$ *that contains it.*

Definition 4. *Given a workflow instance* $(W, D, \mathcal{I})$*, where* $W = (S, P, \mathcal{D})$*, and a step* $s \in S$*, the sets of input and output data elements of* s *are denoted with* $In^D(s)$ *and* $Out^D(s)$*, respectively, and defined as:*

$$In^D(s) = \{d \mid (d, p) \in \mathcal{I} \ \wedge \ p \in In(s)\} \qquad Out^D(s) = \{d \mid (d, p) \in \mathcal{I} \ \wedge \ p \in Out(s)\}$$

Introducing more precise evaluation semantics, triggering strategies, or limitations on the dependencies structure can specialise this general definition to an actual workflow model (e.g., a Petri Net [31] or Coloured Petri Nets [16], a Kahn Processing Network [17], or a Synchronous Dataflow Graph [20]).

2.3 Distributed Workflow Models

A *distributed workflow* is a workflow whose steps can target different *deployment locations* in charge of executing them. To compute the step, the corresponding location must have access to or store all the input data elements, additionally, it will store all the output data elements on its local scope. Locations can be *heterogeneous*, exposing different hardware devices, software libraries, and security levels. Consequently, the steps are explicitly mapped onto execution locations depending on their computing requests. Given that, a distributed workflow model must contain a specification of the workflow structure, the set of available locations, and a mapping relation between steps and locations.

Definition 5. *A distributed workflow is a tuple* $(W, L, \mathcal{M})$*, where* $W = (S, P, \mathcal{D})$ *is a workflow,* L *is the set of available locations, and* $\mathcal{M} \subseteq (S \times L)$ *is a mapping relation stating which locations are in charge of executing each workflow step.*

Each location can execute multiple steps on it, and a single step can be mapped onto multiple locations. Multiple steps related to a single location introduce a *temporal constraint*: all the involved steps compete to acquire the location's resources. They can be serialised if the location does not have enough resources to execute all of them concurrently. Conversely, multiple locations related to a single step express a *spatial constraint*: all involved locations must collaborate to execute the step. This work does not impose any particular strategy for scheduling different step executions on a single location when temporal constraints arise. However, it is helpful to know the *work queue* of a given location l, i.e., the set of steps mapped onto it.

Definition 6. *Given a distributed workflow* $(W, L, \mathcal{M})$*, where* $W = (S, P, \mathcal{D})$*, and a location* $l \in L$*, the set of steps mapped onto* l *is called the work queue of* l*, denoted as* $Q(l)$ *and defined as:* $Q(l) = \{s \mid l \in \mathcal{M}(s)\}$.

Similarly to what was discussed in Sect. 2.2, a single distributed workflow model can generate potentially infinite *distributed workflow instances* with different data elements and condition evaluations.

Definition 7. *A distributed workflow instance is a tuple* $\mathtt{I} = (W, L, \mathcal{M}, D, \mathcal{I})$ *where* $(W, L, \mathcal{M})$ *is a distributed workflow,* D *is a set of data elements, and* $\mathcal{I} \subseteq (D \times P)$ *is a mapping relation connecting each data element* $d \in D$ *to the port* $p \in P$ *that contains it.*

Example 1. Fig. 1 shows an example of a distributed workflow model. A step s_1 produces two different output data elements d_1 and d_2, which are mapped to ports p_1 and p_2. The second and the third step, s_2 and s_3 depend on the data elements on the ports p_1 and p_2, respectively. None of them produces other outputs. This workflow is mapped onto four locations. Step s_1 is executed on location l_d, while s_2 is offloaded to l_1 and step s_3 is mapped to two locations l_2 and l_3. Using definitions above, Fig. 1 can be written as follows:

$$W = (\{s_1, s_2, s_3\}, \{p_1, p_2\}, \{(s_1, p_1), (s_1, p_2), (p_1, s_2), (p_2, s_3)\})$$
$$L = \{l_d, l_1, l_2, l_3\} \qquad \mathcal{M} = \{(s_1, l_d), (s_2, l_1), (s_3, l_2), (s_3, l_3)\}$$
$$D = \{d_1, d_2\} \qquad \mathcal{I} = \{(d_1, p_1), (d_2, p_2)\}$$

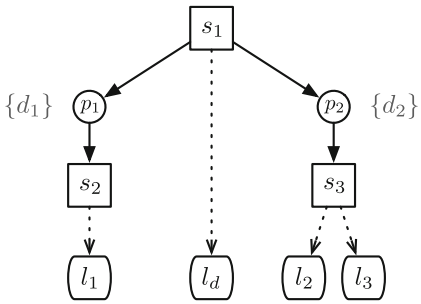

Fig. 1. Example of a distributed workflow model. Steps are represented as squares and ports as circles. Dependency links between steps and ports are depicted as arrows with black-filled heads. Locations are represented as squashed rectangles. Mapping relations are expressed as dotted arrows. A potential instance of this model can be derived by adding data elements, denoted as sets of values, near their related port.

3 The SWIRL Representation

This section introduces SWIRL, a "Scientific Workflow Intermediate Representation Language". Given a distributed workflow instance (Sect. 2.3), SWIRL can model a decentralised execution plan, called *workflow system*, by inferring and projecting execution traces on each involved location and specifying

$$(\text{Id}_{|}) \quad e \mid \mathbf{0} \equiv e \qquad (\text{Id}_{\cdot}) \quad \mathbf{0}.e \equiv e \wedge e.\mathbf{0} \equiv e \qquad (\text{Comt}_u) \quad u \mid u' \equiv u' \mid u \ , \ u \in e, \mathtt{W}$$

Fig. 2. SWIRL structural congruence rules.

$$(\text{Exec}) \quad \frac{\forall l_i \in \mathcal{M}(s) \quad \wedge \quad In^D(s) \subseteq D_i}{\prod_{i,l_i \in \mathcal{M}(s)} \langle l_i, D_i, \mathtt{exec}(s, F(s), \mathcal{M}(s)).e_i \rangle \twoheadrightarrow \prod_{i,l_i \in \mathcal{M}(s)} \left\langle l_i, D_i \cup Out^D(s), e_i \right\rangle}$$

$$(\text{L-Comm}) \quad \frac{d \in D}{\left\langle l, D, \mathtt{send}(d \rightarrowtail p, l, l).e \mid \mathtt{recv}(p, l, l).e' \right\rangle \twoheadrightarrow \langle l, D, e \mid e' \rangle}$$

$$(\text{Comm}) \quad \frac{d \in D}{\left\langle l, D, \mathtt{send}(d \rightarrowtail p, l, l').e \right\rangle \mid \left\langle l', D', \mathtt{recv}(p, l, l').e' \right\rangle \twoheadrightarrow \langle l, D, e \rangle \mid \left\langle l', D' \cup \{d\}, e' \right\rangle}$$

$$(\text{L-Par}) \quad \frac{\langle l, D, e_1 \rangle \twoheadrightarrow \langle l, D', e_1' \rangle}{\langle l, D, e_1 \mid e_2 \rangle \twoheadrightarrow \langle l, D', e_1' \mid e_2 \rangle} \qquad (\text{Sec}) \quad \frac{\langle l, D, e_1 \rangle \twoheadrightarrow \langle l, D', e_1' \rangle}{\langle l, D, e_1.e_2 \rangle \twoheadrightarrow \langle l, D', e_1'.e_2 \rangle}$$

$$(\text{Par}) \quad \frac{\mathtt{W}_1 \twoheadrightarrow \mathtt{W}_1'}{\mathtt{W}_1 \mid \mathtt{W}_2 \twoheadrightarrow \mathtt{W}_1' \mid \mathtt{W}_2} \qquad (\text{Congr}) \quad \frac{\mathtt{W}_1' \equiv \mathtt{W}_1 \twoheadrightarrow \mathtt{W}_2 \equiv \mathtt{W}_2'}{\mathtt{W}_1' \twoheadrightarrow \mathtt{W}_2'}$$

Fig. 3. SWIRL reduction semantics rules.

inter-location communications using send/receive primitives. The following sections introduce SWIRL syntax and semantics and derive a procedure to formally encode a workflow instance `I` into a SWIRL workflow system `W`.

SWIRL models a distributed execution plan as a workflow system `W`, which can be seen as a parallel composition of *location configurations*, tuples $\langle l, D, e \rangle$, containing the location name l, the set D of data elements laying on l at a given time, and the execution trace e representing the actions to be executed on l.

Definition 8. *The syntax of a workflow system* `W` *is defined by the following grammar:*

$$\begin{aligned} \mathtt{W} &::= \langle l, D, e \rangle \parallel (\mathtt{W}_1 \mid \mathtt{W}_2) \\ e &::= \mu \parallel e_1.e_2 \parallel (e_1 \mid e_2) \parallel \mathbf{0} \\ \mu &::= \mathtt{exec}(s, F(s), \mathcal{M}(s)) \parallel \mathtt{send}(d \rightarrowtail p, l, l') \parallel \mathtt{recv}(p, l, l') \\ F(s) &::= In^D(s) \mapsto Out^D(s) \end{aligned}$$

Each execution trace e is constructed from the predicates μ, which can be composed using two operators: the *sequential execution* $e_1.e_2$ and the *parallel composition* $e_1 \mid e_2$. The $\mathbf{0}$ symbol represents the empty trace.

A predicate μ represents an action to be performed during workflow execution. Predicates $\mathtt{send}(d \rightarrowtail p, l, l')$ and $\mathtt{recv}(p, l, l')$ allow transferring the data element d over port p from location l to location l'. Modelling ports and data separately seams redundant, but we prefer to keep them divided for the future extensions of the framework, as adding the loops. The $\mathtt{exec}(s, F(s), \mathcal{M}(s))$ action represents the execution of step s. Besides the name of the step, this predicate contains the set $\mathcal{M}(s)$ of locations onto which s is mapped and the *dataflow*

$F(s)$, i.e., the set $In^D(s)$ of input data needed by s and the set $Out^D(s)$ of output data produced on each $l \in \mathcal{M}(s)$ after the execution of s.

3.1 Semantics

The SWIRL semantics is defined in terms of a *reduction semantics*.

Definition 9. *The SWIRL semantics is defined by the reduction relation* $\longrightarrow$ *defined as a smallest relation closed under the rules of Figs. 2 and 3.*

The structural congruence properties are reported in Fig. 2. The commutativity of the parallel composition in location and the execution trace level is defined with rule (Comt$_u$). For both operators, parallel composition and sequential execution, the identity element is $\mathbf{0}$ (rules (Id$_|$) and (Id$_.$)).

The rules of a SWIRL semantics are depicted in Fig. 3. The step execution is performed by the (Exec) rule. It collects all the locations $\mathcal{M}(s)$ onto which step s is mapped and synchronises the execution action. The data $Out^D(s)$ produced by the step execution are added to the set D_i in all executing locations. Rule (L-Comm) describes local communication, while rule (Comm) represents a data transfer between two locations. In the latter case, the involved data element is *copied* to the targeted (receiving) location. Note that communications do not consume the data element on the sending location.

Assuming that configuration $\langle l, D, e_1 \rangle$ can be computed, rules (L-Par) and (Seq) allow for the execution of the parallel and the sequential composition inside the same location, respectively. The execution of the workflow sub-system as a part of a larger system is allowed by the rule (Par). The (Congr) rule allows the application of structural congruence.

When a step s is mapped onto multiple locations, each of them must contain an `exec` predicate with the set of involved locations. Such predicates introduce *synchronisation points* among different locations, as all involved execution traces must step forward in a single pass. Additionally, each location must have a copy of the input data $In^D(s)$, requiring multiple `send` operations for each element $d \in In^D(s)$, and will own a copy of $Out^D(s)$.

Example 2. The behaviour of the distributed workflow instance given in Fig. 1 can be modelled as a workflow system $\mathtt{W}$ with the following syntax:

$$\mathtt{W} = \langle l_d, \emptyset, e_d \rangle \mid \prod_{i=1}^{3} \langle l_i, \emptyset, e_i \rangle$$

$$\begin{aligned}
e_d &= \mathtt{exec}(s_1, \emptyset \mapsto \{d_1, d_2\}, \{l_d\}).\big(\mathtt{send}(d_1 \rightarrowtail p_1, l_d, l_1) \mid \\
&\qquad \mathtt{send}(d_2 \rightarrowtail p_2, l_d, l_2) \mid \mathtt{send}(d_2 \rightarrowtail p_2, l_d, l_3)\big) \\
e_1 &= \mathtt{recv}(p_1, l_d, l_1).\mathtt{exec}(s_2, \{d_1\} \mapsto \emptyset, \{l_1\}) \\
e_2 &= \mathtt{recv}(p_2, l_d, l_2).\mathtt{exec}(s_3, \{d_2\} \mapsto \emptyset, \{l_2, l_3\}) \\
e_3 &= \mathtt{recv}(p_2, l_d, l_3).\mathtt{exec}(s_3, \{d_2\} \mapsto \emptyset, \{l_2, l_3\})
\end{aligned}$$

In the execution trace e_d, step s_1 is sending output data d_2 to both locations $l_2, l_3 \in \mathcal{M}(s_3)$ through the same port p_2.

3.2 Workflow Model Encoding

Example 2 describes the encoding of a distributed workflow instance $\mathtt{I}$ into a workflow system $\mathtt{W}$. This section introduces a formal methodology to perform this encoding automatically for any distributed workflow instance.

In SWIRL, the execution trace e_l of a location $l \in L$ models the actions required to execute all the steps in its work queue $Q(l)$. In this respect, e_l can be seen as the parallel composition of *building blocks* $B_l(s)$, one for each $s \in Q(l)$. Each building block $B_l(s)$ contains the same sequence of actions: (i) receives all the necessary data elements for the step execution in which case it is necessary to determine all input data elements ($In^D(s)$) and for each element to identify the step producing it ($In(\mathcal{I}(d_i))$) and the locations on which the steps are mapped to ($\mathcal{M}(In(\mathcal{I}(d_i)))$) (ii) executes the step s; (iii) sends the produced data elements ($Out^D(s)$) to the locations onto which the receiving steps are mapped (one data element can be sent, over the same port, to the different steps/locations, therefore it is necessary to identify the steps data d_i is sent to with $Out(\mathcal{I}(d_i))$ and the locations l_j on which each step is deployed).

Definition 10. *Given a distributed workflow instance* $\mathtt{I} = (W, L, \mathcal{M}, D, \mathcal{I})$*, a deployment location* $l \in L$ *and a step* $s \in S$ *s.t.* $l \in \mathcal{M}(s)$*, the building block representing* s *in* e_l *is denoted by* $B_l(s)$ *and defined as:*

$$B_l(s) = \left(\prod_{d_i \in In^D(s)} \prod_{l_j \in \mathcal{M}(In(\mathcal{I}(d_i)))} \mathtt{recv}(\mathcal{I}(d_i), l_j, l) \right).$$
$$\mathtt{exec}(s, In^D(s) \mapsto Out^D(s), \mathcal{M}(s)).$$
$$\left(\prod_{d_i \in Out^D(s)} \prod_{s_k \in Out(\mathcal{I}(d_i))} \prod_{l_j \in \mathcal{M}(s_k)} \mathtt{send}(d_i \rightarrowtail \mathcal{I}(d_i), l, l_j) \right)$$

Definition 10 introduces the general form of $B_l(s)$, which holds for steps connected to both input and output ports. If a step does not consume input data, as in the case of step s_1 from Example 2, the receiving part of $B_l(s)$ is modelled with $\mathbf{0}$, resulting in $B_{l_d}(s_1) = \mathbf{0}.\mathtt{exec}(s, \emptyset \mapsto \{d_1\}, \{l_d\}).\mathtt{send}(d_1 \rightarrowtail p_1, l_d, l_1)$. The same applies to steps that do not produce output data.

The concept of building blocks $B_l(s)$ allows for the modular construction of execution traces by processing one pair (s, l) at a time. Intuitively, for each mapping pair step-location (s, l), corresponding building blocks $B_l(s)$ are made and added to the execution trace of the location l. Another important information is the *instance data distribution* on the locations, denoted by $G(l) = \{d | d \in D_l\}$.

Definition 11. *The encoding function* $[\![\cdot]\!] : \mathcal{W}_\mathtt{I} \rightarrow \mathcal{W}_\mathtt{W}$*, where* $\mathcal{W}_\mathtt{I}$ *and* $\mathcal{W}_\mathtt{W}$ *are the sets of distributed workflow instances and workflow systems represented in SWIRL, respectively, is inductively defined as follows:*

$$[\![\mathtt{I}]\!] = [\![\mathtt{I}, \mathcal{M}, G; \mathtt{W}]\!] \quad \textit{where } \mathtt{W} = \prod_{\forall l_i \in L} \langle l_i, \emptyset, e_l \rangle$$

$$[\![\mathtt{I}, \mathcal{M} \cup (s,l), G; \mathtt{W} \mid \langle l, \emptyset, e_l \rangle]\!] = [\![\mathtt{I}, \mathcal{M}, G; \mathtt{W} \mid \langle l, \emptyset, e_l \mid B_l(s) \rangle]\!]$$

$$[\![\mathtt{I}, \mathcal{M}, G \cup G(l); \mathtt{W} \mid \langle l, \emptyset, e_l \rangle]\!] = [\![\mathtt{I}, \mathcal{M}, G; \mathtt{W} \mid \langle l, G(l), e_l \rangle]\!]$$

$$[\![\mathtt{I}, \emptyset, \emptyset; \mathtt{W}]\!] = \mathtt{W}$$

Formally, the encoding operator can be defined as a function with four input parameters: (i) a workflow instance $\mathtt{I}$ to be translated; (ii) the set of pairs (s, l) containing all mappings in $\mathcal{M}$; (iii) the set G representing the distribution of the data over locations; (iv) placeholder to build the workflow system $\mathtt{W}$. The translation starts by adding the auxiliary parameters into the encoding process and inside a SWIRL placeholder, creating a workflow containing locations configurations for each location in the workflow instance $\mathtt{I}$ (for all $l \in L$). The iteration process is divided in two phases, first at each iteration, the encoding function takes the pair (s, l), identify the location l and add the building block $B_l(s)$ into the execution trace to be executed on the location l. When all pairs are encoded, in the second phase, the iteration is on the distribution of the data over locations. Each set $G(l) \subseteq G$ is encoded to the corresponding locations. In that way, the trace e_l is the parallel composition of building blocks $B_l(s)$ for each $s \in Q(l)$. The encoding finishes when both sets $\mathcal{M}$ and G are empty.

3.3 Consistency of SWIRL Semantics

This section defines a concurrency relation on the derivations of a workflow system $\mathtt{W}$, which is then used to show the consistency of different execution diagrams through the semantics. As commonly done in the literature, this section only considers reachable workflow systems defined below.

Definition 12. *Given a distributed workflow instance* $\mathtt{I} = (W, L, \mathcal{M}, D, \mathcal{I})$ *and the function* $[\![\cdot]\!] : \mathcal{W}_{\mathtt{I}} \to \mathcal{W}_{\mathtt{W}}$*, the* initial state *of a distributed workflow system is*

$$\mathtt{W}_{Init} = [\![\mathtt{I}, L, *]\!] = \prod_{l_j \in L} \left\langle l_j, \emptyset, \prod_{s \in Q(l_j)} B(s) \right\rangle$$

Definition 13. *A state of a workflow system* $\mathtt{W}$ *is* reachable *if it can be derived from the initial state* ($\mathtt{W}_{Init}$) *by applying the rules in Figs. 2 and 3.*

Having a transition $t : \mathtt{W} \longrightarrow \mathtt{W}'$, the workflow states $\mathtt{W}$ and $\mathtt{W}'$ are called *source* and *target* of the transition t, respectively. The *concurrency relation* is defined on the transitions having the same source. Formally:

Definition 14 (Concurrency relation). *Two different transitions* $t_1 : \mathtt{W} \longrightarrow \mathtt{W}_1$ *and* $t_2 : \mathtt{W} \longrightarrow \mathtt{W}_2$ *having the same source, are always concurrent, written* $t_1 \smile t_2$.

Following the standard notation, let t_2/t_1 represent a transition t_2 executed after the transition t_1. The concurrency relation is used to prove the *Church-Rosser property*, which states that when two concurrent transitions execute at the same time, the ordering of the executions does not impact the eventual result. This finding shows that the concurrent semantics is confluent. Formally:

Lemma 1 (Church-Rosser property). *Given two concurrent transitions* $t_1 : \mathtt{W} \longrightarrow \mathtt{W}_1$ *and* $t_2 : \mathtt{W} \longrightarrow \mathtt{W}_2$*, there exist two transitions* $t_2/t_1 : \mathtt{W}_1 \longrightarrow \mathtt{W}_3$ *and* $t_1/t_2 : \mathtt{W}_2 \longrightarrow \mathtt{W}_3$ *having the same target.*

4 Optimisation

This section introduces an *optimisation function* that scans the entire workflow system to remove redundant communications, improving performance. In particular, there are two cases in which execution traces can be optimised: (*i*) communications between steps deployed on the same location, which are always redundant; (*ii*) multiple communications of the same data element between a pair of locations, when different steps mapped onto the destination location require the same input data from the same ports.

The encoding function adds a communication to the workflow system $\mathtt{W}$ every time a data element is required for the execution of a step, no matter if it is already present at the destination location, creating unnecessary communications. For instance, consider a location $\langle l, D, e \rangle$ where $D = \emptyset$ and

$$e = \mathtt{recv}(p, l_1, l).\mathtt{exec}(s, \{d\} \mapsto \{d_1\}, \{l\}).\mathtt{send}(d_1 \rightarrowtail p_1, l, l) \mid \\ \mathtt{recv}(p_1, l, l).\mathtt{exec}(s_1, \{d_1\} \mapsto \emptyset, \{l\})$$

After the execution of the step s, the data element d_1 is saved on the location l ($D \cup \{d_1\}$), therefore the $\mathtt{send}/\mathtt{recv}$ pair does not affect the state of $\mathtt{W}$. By removing the unnecessary communication, the trace e can be rewritten as:

$$e' = \mathtt{recv}(p, l_1, l).\mathtt{exec}(s, \{d\} \mapsto \{d_1\}, \{l\}) \mid \mathtt{exec}(s_1, \{d_1\} \mapsto \emptyset, \{l\})$$

The rule (Exec) in Fig. 3, preserves dependency between steps s and s_1 by ensuring that step s_1 will not execute until the required data d_1 is produced.

The second optimisation step is to remove redundant communications between different pairs of locations when the same data element is sent multiple times through the same port. For instance, consider two locations $\langle l, D, e \rangle$ and $\langle l', D', e' \rangle$ where $D = D' = \emptyset$ and

$$e = \mathtt{recv}(p, l_1, l).\mathtt{exec}(s, \{d\} \mapsto \{d_1\}, \{l\}).(\prod_{i=1}^{3} \mathtt{send}(d_1 \rightarrowtail p_1, l, l'))$$

$$e' = \prod_{i=1}^{3} \mathtt{recv}(p_1, l, l').\mathtt{exec}(s_i, \{d_1\} \mapsto \emptyset, \{l'\})$$

The first location l sends the data element d_1 to three steps mapped onto location l'. Transferring the data element only once is enough, as the subsequent communications will not affect the state of $\mathtt{W}$, hence, there is:

$$e = \mathtt{recv}(p, l_1, l).\mathtt{exec}(s, \{d\} \mapsto \{d_1\}, \{l\}).\mathtt{send}(d_1 \rightarrowtail p_1, l, l')$$

$$e' = \mathtt{recv}(p_1, l, l').\mathtt{exec}(s_k, \{d_1\} \mapsto \emptyset, \{l'\}) \mid \prod_{i=1, i \neq k}^{3} \mathtt{exec}(s_i, \{d_1\} \mapsto \emptyset, \{l'\})$$

The optimisation of a workflow system $\mathtt{W}$ is defined in terms of three functions: the first and the second[1] ones start the optimisation process and controls it till the end, by taking the additional parameter A (the set of all prefixes/actions) and calling the third function that actually rewrite the execution trace of a location. It goes through the execution traces of the workflow $\mathtt{W}$ and breaks them into single action (prefix) blocks. Analysing the blocks one by one, it performs the following actions: (*i*) if the predicate is a part of the communication on the same location, it is removed; (*ii*) if the predicate is already in the set A, it is removed as well (meaning the same data element was already sent to the same location through the same port, just to different step); (*iii*) otherwise, the predicate is added to the set A and the drilling function moves to the next element.

Definition 15. *Given the workflow system* $\mathtt{W}$ *and sets of workflow and optimised systems* $\mathcal{W}_{\mathtt{W}}$ *and* $\mathcal{W}_{\mathtt{O}}$*, respectively, the optimisation function* $\mathtt{W}$, $[\![\cdot]\!] : \mathcal{W}_{\mathtt{W}} \to \mathcal{W}_{\mathtt{O}}$ *is defined in terms of the auxiliary functions,* $[\![\cdot]\!] : \mathcal{W}_{\mathtt{W}} \times A \to \mathcal{W}_{\mathtt{O}}$ *and* $\wr\cdot\int : \mathcal{W}_{\mathtt{W}} \to \mathcal{W}_{\mathtt{O}}$ *(where* $\mathtt{o} \in \{|, .\}$ *and* $A_{l,l} = \{\mathtt{send}(d \rightarrowtail p, l, l), \mathtt{recv}(p, l, l)\}$*) as follows:*

$$
\begin{aligned}
[\![\mathtt{W}]\!] &= [\![\wr \mathtt{W} \int, \emptyset]\!] \\
[\![\wr \langle l, D, e \rangle \int, A]\!] &= \langle l, D, [\![\wr e \int, A]\!] \rangle \\
[\![\wr \mathtt{W}_1 \mid \mathtt{W}_2 \int, A]\!] &= [\![\wr \mathtt{W}_1 \int, A]\!] \mid [\![\wr \mathtt{W}_2 \int, A]\!] \\
\wr e \mathtt{\ o\ } e_1 \int &= \wr e \int \mathtt{\ o\ } \wr\ e_1 \int \\
[\![e \mathtt{\ o\ } \wr\ \mu \int \mathtt{\ o\ } \wr\ e_1 \int, A]\!] &= \begin{cases} [\![e \mathtt{\ o\ } \mathbf{0} \mathtt{\ o\ } \wr\ e_1 \int, A]\!] & \textit{if } \mu \in A \quad \vee \quad \mu \in A_{l,l} \\ [\![e \mathtt{\ o\ } \mu \mathtt{\ o\ } \wr\ e_1 \int, A \cup \mu]\!] & \textit{otherwise} \end{cases} \\
[\![e, A]\!] &= e
\end{aligned}
$$

The two workflow systems $\mathtt{W}$ and $\mathtt{O} = [\![\mathtt{W}]\!]$ are modelling the same behaviour of the distributed workflow system, i.e. the computations of the workflow steps are executed in the same order in both systems with the difference in the number of communications. Therefore, the weak barbed bisimulation [33] is used to define the relation between the distributed workflow system and its optimised version.

To highlight that the executing action is a communication, it is labelled by τ. Therefore, $\mathtt{W} \xrightarrow{\tau} \mathtt{W}'$ indicates that workflow system $\mathtt{W}$ can evolve into $\mathtt{W}'$ by performing the communication (transfer) action. The reflexive and transitive closure of $\xrightarrow{\tau}$ is denoted with $\stackrel{\tau}{\Rightarrow}$ and the transition $\mathtt{W} \stackrel{\tau}{\Rightarrow} \mathtt{W}'$ express the ability of the system $\mathtt{W}$ to evolve into $\mathtt{W}'$ by executing some number, possibly zero, of τ actions (communications). Given the transition $\mathtt{W} \to \mathtt{W}'$ (any type of action, including the communication), if the same action can be executed after a certain number of communication actions, it is denoted as $\mathtt{W} \stackrel{\tau}{\Rightarrow} \to \mathtt{W}'$.

The observable elements in this setting are the executions of the steps and it is denoted by $\mathtt{W} \downarrow_\nu$ (resp. $\mathtt{O} \downarrow_\nu$) where $\nu = \mathtt{exec}(s, F(s), \mathcal{M}(s))$ where the

[1] The two functions have the same notation to simplify the notation, they can be easily distinguished because of the different number of arguments.

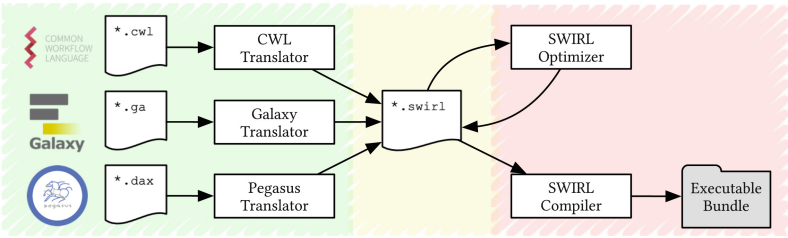

Fig. 4. The SWIRL compiler toolchain.

weak barb is denoted by $\mathtt{W} \Downarrow_\nu$ (resp. $\mathtt{0} \Downarrow_\nu$), and it is defined as $\mathtt{W} \stackrel{\tau}{\Rightarrow}\downarrow_\nu$ (resp. $\mathtt{0} \stackrel{\tau}{\Rightarrow}\downarrow_\nu$). Hence, the barbed bisimulation will check that all the step executions in a workflow system can be matched by the executions in the optimised one.

Definition 16. *A relation $\mathcal{R} \subseteq \mathcal{W} \times \mathcal{O}$ is a weak barbed simulation if $\mathtt{W}\mathcal{R}[\![\mathtt{W}]\!]$:*

- $\mathtt{W} \downarrow_\nu$ *implies* $[\![\mathtt{W}]\!] \Downarrow_\nu$
- $\mathtt{W} \longrightarrow \mathtt{W}'$ *implies* $[\![\mathtt{W}]\!] \Rightarrow [\![\mathtt{W}']\!]$ *with* $\mathtt{W}'\mathcal{R}[\![\mathtt{W}']\!]$

A relation $\mathcal{R} \subseteq \mathcal{W} \times \mathcal{O}$ is a weak barbed bisimulation if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are weak barbed simulations. Weak bisimilarity, $\approx$, is the largest weak barbed bisimulation.

The next theorem shows the operational correspondence between a distributed workflow system and its optimised term.

Theorem 1. *For any distributed workflow system* $\mathtt{W}$, $\mathtt{W} \approx [\![\mathtt{W}]\!]$.

5 Implementation

The SWIRL compiler reference implementation[2], called `swirlc`, follows the same line as the theoretical approach, separating scientific workflows' design and runtime phases. On the one hand, it allows the translation of high-level, product-specific workflow languages designed for direct human interaction to chains of low-level primitives easily understood by distributed runtime systems. A common representation fosters composability and interoperability among different workflow models, which can be easily combined into a single workflow system. Moreover, the translation process is performed with the formal consistency guarantees discussed in Sect. 3.2.

Finally, a SWIRL-based compiler can translate a workflow system $\mathtt{W}$ to a high-performance, self-contained workflow execution bundle based on send/receive communication protocols and runtime libraries, which can easily be included in a Research Object [5], improving reproducibility. An advanced compiler can also generate multiple execution bundles from the same workflow system, each optimised for a different execution environment (e.g., Cloud, HPC, or Edge),

[2] https://github.com/alpha-unito/swirlc.

improving performance. As a bonus feature, the intrinsically distributed nature of SWIRL execution traces promotes decentralised runtime architectures, avoiding the single point of failure introduced by a centralised control plane.

Figure 4 sketches the SWIRL compiler toolchain. We implemented the SWIRL grammar using ANTLR [27], and we automatically generated Python3 parser classes to process the SWIRL syntax. All the components of the SWIRL toolchain rely on these parsers to process `*.swirl` files. An abstract `SWIRLTranslator` class implements the encoding function, producing a SWIRL file from a workflow instance `I`. A concrete implementation specialises the `SWIRLTranslator` logic to the semantics of a given workflow language, e.g., CWL

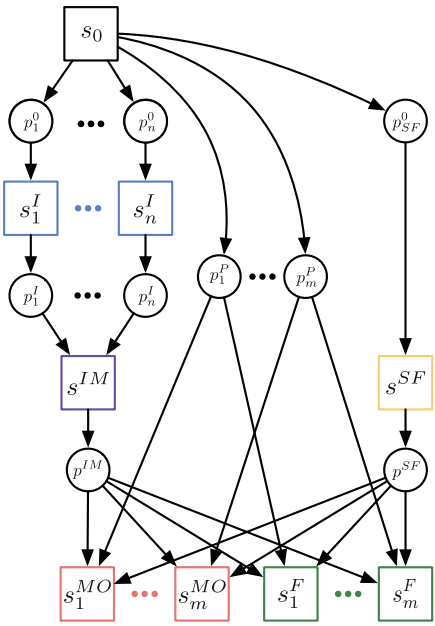

Fig. 5. Graphical representation of the 1000 Genomes workflow contains five classes of steps mapped to diverse locations: (*i*) `individuals` (blue), number of steps n, mapped to locations l_j^I, $j \in [1, a]$; (*ii*) `individuals_merge` (violet), a single step mapped to location l^{IM}; (*iii*) `sifting` (yellow), a single step mapped to location l^{IM}; (*iv*) `mutations_overlap` (red), number of steps m, mapped to locations l_t^{MO}, $t \in [1, b]$, and (*v*) `frequency` (green) number of steps m, mapped to locations l_k^F, $k \in [1, c]$. The initial step s_0, mapped to driver location l_d is a step that sends each input data element to the correct location for processing. The mapping between data elements and ports, where $i \in [1, n]$ and $h \in [1, m]$, is: $\mathcal{I} = \left\{ \begin{array}{l} (d_i^0, p_i^0), (d_h^P, p_h^P), (d_{SF}^0, p_{SF}^0), \\ (d_i^I, p_i^I), (d^{IM}, p^{IM}), (d^{SF}, p^{SF}) \end{array} \right\}$ (Color figure online)

[11], DAX (for Pegasus [12]), or the Galaxy Workflow Format (GWF) [40][3]. A `SWIRLOptimizer`[5] class implements the optimisation function $[\![\cdot]\!] : \mathcal{W}_{\mathtt{W}} \to \mathcal{W}_{\mathtt{O}}$, generating an optimised $*$`.swirl` file. Finally, an abstract `SWIRLCompiler` class produces an executable bundle from a $*$`.swirl` file and a declarative metadata file, which contains additional information not currently modelled in the SWIRL semantics, e.g., step commands, data types and location IP addresses. We have implemented a simple compiler class that generates a multithreaded Python program for each location, relying on TCP sockets for send/receive communications.

6 Evaluation

This section tests the flexibility of the SWIRL representation on the 1000 Genomes workflow [35], a Bioinformatics pipeline aiming at fetching, parsing and analysing data from the 1000 Genomes Project [39] to identify mutational overlaps and provide a null distribution for rigorous statistical evaluation of potential disease-related mutations. However, we used the 1000 Genomes applications written in C++ [22]. Figure 5 shows a slightly simplified version of the 1000 Genomes workflow model. We removed some ports to simplify the notation, but their absence does not affect the reasoning reported in the rest of this Section. Note that the number of locations could be smaller than the number of steps. Hence, there could be a case when more steps are mapped to the same location.

The corresponding workflow system $\mathtt{W}$ can be constructed using the encoding function $[\![\cdot]\!] : \mathcal{W}_{\mathtt{I}} \to \mathcal{W}_{\mathtt{W}}$ (Sect. 3.2). It can be written as follows:

$$\mathtt{W} = \prod_{i \in \{d,SF,IM\}} \left\langle l^i, \emptyset, e^i \right\rangle \mid \prod_{j=1}^{a} \left\langle l^I_j, \emptyset, e^I_j \right\rangle \mid \prod_{t=1}^{b} \left\langle l^{MO}_t, \emptyset, e^{MO}_t \right\rangle \mid \prod_{k=1}^{c} \left\langle l^F_k, \emptyset, e^F_k \right\rangle$$

where each execution trace e^*_* defines the actions (steps and data transfers) depicted in Fig. 5 to be executed on the corresponding location. For instance, if the driver location is taken, the execution trace e^d is defines as:

$$e^d = \prod_{i=1}^{n} \mathtt{send}(d^0_i \rightarrowtail p^0_i, l^d, l^I_j) \mid \mathtt{send}(d^0_{SF} \rightarrowtail p^0_{SF}, l^d, l^{SF}) \mid \prod_{h=1}^{m} (\mathtt{send}(d^P_h \rightarrowtail p^P_h, l^d, l^{MO}_t) \mid \mathtt{send}(d^P_h \rightarrowtail p^P_h, l^d, l^F_k))$$

The full representation of $\mathtt{W}$ is discussed in [10]. The 1000 Genomes workflow modelled above can be reproduced using the SWIRL implementation (Sect. 5). To keep the experiment small and ease reproducibility, the ten homogeneous execution locations and a single chromosome, i.e., a single workflow instance, are considered. The necessary installing package and instructions on how to run the experiment can be find in [9].

[3] The implementation of the CWL and GWF translators and the `SWIRLOptimizer` are ongoing works.

7 Conclusion

This work introduced SWIRL, a "Scientific Workflow Intermediate Representation Language" based on send/receive communication primitives. An encoding function maps any workflow instance onto a distributed execution plan `W`, fostering interoperability and composability of different workflow models. A set of rewriting rules allows for automatic optimisation of data communications, improving performance with correctness and consistency guarantees. The optimised SWIRL representation can be compiled into one or more self-contained executable bundles addressing specific execution environments, ensuring reproducibility and embracing heterogeneity. SWIRL already proved itself to be flexible enough to model a real large-scale scientific workflow (even if still not supporting all features of modern WMSs).

The foundational contribution of SWIRL is to propose a novel direction to solve well-known problems in the field of scientific workflows. Indeed, SWIRL shifts the focus from high-level workflow languages, designed either for direct human interaction or to encode complex, product-specific features, to a low-level minimalistic representation of a workflow execution plan, which is far more manageable from both formalisation methods and compiler toolchains. In this context, we hope that SWIRL can pave the way to a novel, more formal approach to distributed workflow orchestration research.

The formal SWIRL representation gives the possibility to build the formally correct extensions, for instance, adding a type system where the multiparty sessions are enriched with security levels for messages (data in our case) [6] or deriving the causal-consistent reversible framework by applying the approach [19], that later can be used as a base to build fault-tolerance mechanism.

Acknowledgments. This work was supported by: the Spoke 1 "FutureHPC & BigData" of ICSC - Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing, funded by European Union - NextGenerationEU; the EUPEX EU's Horizon 2020 JTI-EuroHPC research and innovation programme project under grant agreement No 101033975.

Data Availability Statement.. The artifact presented in this article is openly available at https://doi.org/10.5281/zenodo.12523000

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Inf. Syst. **30**(4), 245–275 (2005). https://doi.org/10.1016/j.is.2004.02.002
2. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed Parallel Databases **14**(1), 5–51 (2003). https://doi.org/10.1023/A:1022883727209
3. Amstutz, P., Mikheev, M., Crusoe, M.R., Tijanic, N., Lampa, S., et al.: Existing workflow systems. common workflow language wiki (2022). https://s.apache.org/existing-workflow-systems. Accessed 05 Oct 2023

4. Ayachit, U., Bauer, A.C., Duque, E.P.N., Eisenhauer, G., Ferrier, N.J., et al.: Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016, pp. 921–932. IEEE Computer Society (2016). https://doi.org/10.1109/SC.2016.78
5. Bechhofer, S., Buchan, I.E., Roure, D.D., Missier, P., Ainsworth, J.D., et al.: Why linked data is not enough for scientists. Futur. Gener. Comput. Syst. **29**(2), 599–611 (2013). https://doi.org/10.1016/j.future.2011.08.004
6. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Information flow safety in multiparty sessions. Math. Struct. Comput. Sci. **26**(8), 1352–1394 (2016). https://doi.org/10.1017/S0960129514000619
7. Colonnelli, I., Aldinucci, M., Cantalupo, B., Padovani, L., Rabellino, S., et al.: Distributed workflows with Jupyter. Futur. Gener. Comput. Syst. **128**, 282–298 (2022). https://doi.org/10.1016/j.future.2021.10.007
8. Colonnelli, I., Cantalupo, B., Merelli, I., Aldinucci, M.: StreamFlow: cross-breeding cloud with HPC. IEEE Trans. Emerg. Top. Comput. **9**(4), 1723–1737 (2021). https://doi.org/10.1109/TETC.2020.3019202
9. Colonnelli, I., Medic, D., Mulone, A., Bono, V., Padovani, L., Aldinucci, M.: Artifact for paper "Introducing SWIRL: An Intermediate Representation Language for Scientific Workflows". https://doi.org/10.5281/zenodo.12523000 (2024). Accessed 26 June 2024
10. Colonnelli, I., Medić, D., Mulone, A., Bono, V., Padovani, L., Aldinucci, M.: Introducing swirl: an intermediate representation language for scientific workflows (2024). https://iris.unito.it/handle/2318/1989870
11. Crusoe, M.R., Abeln, S., Iosup, A., Amstutz, P., Chilton, J., et al.: Methods included: standardizing computational reuse and portability with the common workflow language. Commun. ACM (2022). https://doi.org/10.1145/3486897
12. Deelman, E., et al.: The evolution of the Pegasus workflow management software. Comput. Sci. Eng. **21**(4), 22–36 (2019). https://doi.org/10.1109/MCSE.2019.2919690
13. Dong Yang, S.S.Z.: Approach for workflow modeling using π-calculus. J. Zhejiang Univ. Sci. 2003 **4**(6), 643–650 (2003). https://doi.org/10.1631/jzus.2003.0643
14. Fahringer, T., Prodan, R., Duan, R., Hofer, J., Nadeem, F., et al.: ASKALON: A development and grid computing environment for scientific workflows. In: Workflows for e-Science, Scientific Workflows for Grids, pp. 450–471. Springer (2007). https://doi.org/10.1007/978-1-84628-757-2_27
15. Hennessy, M.: A distributed Pi-calculus. Cambridge University Press (2007)
16. Jensen, K.: Coloured petri nets: A high level language for system design and analysis. In: Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings], pp. 342–416 (1989). https://doi.org/10.1007/3-540-53863-1_31
17. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information processing, pp. 471–475. North Holland, Amsterdam, Stockholm, Sweden (1974)
18. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: Information Processing. In: Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977, pp. 993–998. North-Holland (1977)
19. Lanese, I., Medic, D.: A general approach to derive uncontrolled reversible semantics. In: 31st International Conference on Concurrency Theory, CONCUR 2020,

September 1-4, 2020, Vienna, Austria (Virtual Conference). LIPIcs, vol. 171, pp. 33:1–33:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPICS.CONCUR.2020.33
20. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987). https://doi.org/10.1109/PROC.1987.13876
21. Ludäscher, B., et al.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice and Experience **18**(10), 1039–1065 (2006). https://doi.org/10.1002/cpe.994
22. Martinelli, A.R., Torquati, M., Aldinucci, M., Colonnelli, I., Cantalupo, B.: Capio: a middleware for transparent i/o streaming in data-intensive workflows. In: 2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE, Goa, India (2023). https://doi.org/10.1109/HiPC58850.2023.00031
23. Medic, D., Aldinucci, M.: Towards formal model for location aware workflows. In: 47th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2023, Torino, Italy, June 26-30, 2023, pp. 1864–1869. IEEE (2023). https://doi.org/10.1109/COMPSAC57700.2023.00289
24. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
25. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 89), Pacific Grove, California, USA, 5–8 June, 1989, pp. 14–23. IEEE Computer Society (1989). https://doi.org/10.1109/LICS.1989.39155
26. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. **67**(2–3), 162–198 (2007). https://doi.org/10.1016/j.scico.2007.03.002
27. Parr, T.J., Quong, R.W.: ANTLR: a predicated- LL(k) parser generator. Softw. Pract. Exp. **25**(7), 789–810 (1995). https://doi.org/10.1002/spe.4380250705
28. Plankensteiner, K., Montagnat, J., Prodan, R.: IWIR: a language enabling portability across grid workflow systems. In: WORKS'11, Proceedings of the 6th Workshop on Workflows in Support of Large-Scale Science, pp. 97–106. ACM (2011). https://doi.org/10.1145/2110497.2110509
29. Puhlmann, F., Weske, M.: Using the *pi*-calculus for formalizing workflow patterns. In: Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings, vol. 3649, pp. 153–168 (2005). https://doi.org/10.1007/11538394_11
30. Reed, D.A., Gannon, D., Dongarra, J.J.: Reinventing high performance computing: Challenges and opportunities. CoRR abs/2203.02544 (2022). https://doi.org/10.48550/arXiv.2203.02544
31. Reisig, W., Rozenberg, G. (eds.): ACPN 1996. LNCS, vol. 1491. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-65306-6
32. Roy, R.B., Patel, T., Gadepally, V., Tiwari, D.: Mashup: making serverless computing useful for HPC workflows via hybrid execution. In: PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 46–60. ACM (2022). https://doi.org/10.1145/3503221.3508407
33. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
34. Siddiqui, M., Villazón, A., Hofer, J., Fahringer, T.: GLARE: a grid activity registration, deployment and provisioning framework. In: Proceedings of the ACM/IEEE

SC2005 Conference on High Performance Networking and Computing, p. 52 (2005). https://doi.org/10.1109/SC.2005.30
35. da Silva, R.F., Filgueira, R., Deelman, E., Pairo-Castineira, E., Overton, I.M., Atkinson, M.P.: Using simple pid-inspired controllers for online resilient resource management of distributed scientific workflows. Futur. Gener. Comput. Syst. **95**, 615–628 (2019). https://doi.org/10.1016/j.future.2019.01.015
36. Taylor, I.J., Shields, M.S., Wang, I., Harrison, A.: The Triana workflow environment: architecture and applications. In: Workflows for e-Science, Scientific Workflows for Grids, pp. 320–339. Springer (2007). https://doi.org/10.1007/978-1-84628-757-2_20
37. Taylor, I.J., Shields, M.S., Wang, I., Rana, O.F.: Triana applications within grid computing and peer to peer environments. J. Grid Comput. **1**(2), 199–217 (2003). https://doi.org/10.1023/B:GRID.0000024074.63139.ce
38. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the Condor experience. Concurrency and Computation: Practice and Experience **17**(2–4), 323–356 (2005). https://doi.org/10.1002/cpe.938
39. The 1000 Genomes Project Consortium: A global reference for human genetic variation. Nature **526**(7571), 68–74 (2015). https://doi.org/10.1038/nature15393
40. The Galaxy Community: The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2022 update. Nucleic Acids Res. **50**(W1), W345–W351 (2022). https://doi.org/10.1093/nar/gkac247
41. Turi, D., Missier, P., Goble, C.A., Roure, D.D., Oinn, T.: Taverna workflows: Syntax and semantics. In: Third International Conference on e-Science and Grid Computing, e-Science 2007, 10-13 December 2007, Bangalore, India, pp. 441–448. IEEE Computer Society (2007). https://doi.org/10.1109/E-SCIENCE.2007.71
42. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, pp. 15–28. USENIX Association (2012)