Future Generation Computer Systems 157 (2024) 82–97

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Validity constraints for data analysis workflows

Florian Schintke [a,*], Khalid Belhajjame [b], Ninon De Mecquenem [c], David Frantz [d], Vanessa Emanuela Guarino [c,e], Marcus Hilbrich [c], Fabian Lehmann [c], Paolo Missier [f], Rebecca Sattler [c], Jan Arne Sparka [c], Daniel T. Speckhard [c,g], Hermann Stolte [c], Anh Duc Vu [c], Ulf Leser [c]

[a] Zuse Institute Berlin, Takustr. 7, 14195, Berlin, Germany
[b] PSL, Université Paris Dauphine, LAMSADE, Paris, France
[c] Humboldt-Universität zu Berlin, Berlin, Germany
[d] Trier University, Trier, Germany
[e] Max-Delbrück Center for Molecular Medicine, Berlin, Germany
[f] Newcastle University, Newcastle upon Tyne, United Kingdom
[g] Fritz-Haber-Institut der Max–Planck-Gesellschaft, Berlin, Germany

ARTICLE INFO

Keywords:
Scientific workflow systems
Workflow specification languages
Validity constraints
Dependability
Integrity and conformance checking

ABSTRACT

Porting a scientific data analysis workflow (DAW) to a cluster infrastructure, a new software stack, or even only a new dataset with some notably different properties is often challenging. Despite the structured definition of the steps (tasks) and their interdependencies during a complex data analysis in the DAW specification, relevant assumptions may remain unspecified and implicit. Such hidden assumptions often lead to crashing tasks without a reasonable error message, poor performance in general, non-terminating executions, or silent wrong results of the DAW, to name only a few possible consequences. Searching for the causes of such errors and drawbacks in a distributed compute cluster managed by a complex infrastructure stack, where DAWs for large datasets typically are executed, can be tedious and time-consuming.

We propose validity constraints (VCs) as a new concept for DAW languages to alleviate this situation. A VC is a constraint specifying logical conditions that must be fulfilled at certain times for DAW executions to be valid. When defined together with a DAW, VCs help to improve the portability, adaptability, and reusability of DAWs by making implicit assumptions explicit. Once specified, VCs can be controlled automatically by the DAW infrastructure, and violations can lead to meaningful error messages and graceful behavior (e.g., termination or invocation of repair mechanisms). We provide a broad list of possible VCs, classify them along multiple dimensions, and compare them to similar concepts one can find in related fields. We also provide a proof-of-concept implementation for the workflow system Nextflow.

1. Introduction

Data analysis workflows (DAWs, or scientific workflows) are structured descriptions for scientific datasets' scientific analysis [1,2]. DAWs' usage becomes increasingly popular in all scientific domains as datasets grow in size, analyses grow in complexity, and demands grow in terms of speed of development, the throughput of analyses, reusability by others, and reproducibility of results [3–5]. A DAW essentially is a program consisting of individual tasks (programs themselves) with their specific inputs and outputs and a specification of the dependencies between tasks. Executing a DAW means scheduling its tasks on the available computational infrastructure in an order compatible with the data dependencies under some optimization constraints, such as minimal time-to-finish [6–8]. When DAWs are applied for the analysis of large datasets and are executed on clusters of distributed compute nodes, managing their execution also involves resource management, coordination of distributed computations, and file handling [9]. Such distributed executions typically rely on the availability of an infrastructure stack consisting of several components such as (distributed) file systems, resource managers, container managers, and runtime monitoring tools [10].

The interfaces and functionality of these components are not standardized and vary substantially between different systems [10]. Therefore, DAW developers often optimize their code to the used infrastructure (e.g., to the number and memory sizes of available compute nodes)

* Corresponding author.
E-mail address: schintke@zib.de (F. Schintke).

https://doi.org/10.1016/j.future.2024.03.037
Received 6 October 2023; Received in revised form 12 February 2024; Accepted 22 March 2024
Available online 25 March 2024
0167-739X/© 2024 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

and to the particular datasets they wish to analyze (e.g., by hard-coding the number of data partitions for concurrent execution). Furthermore, many tasks of typical DAWs have been written by third parties, providing their specific functionality in a highly optimized manner, while others provide merely ‘glue’ code for data transformation and filtering between original tasks [11,12]. As a result, real-life DAWs are rather brittle artifacts. They are tightly bound to the infrastructure used during development, suffer from intricacies of the programs they embrace, and only work flawless for a narrow range of inputs. Changes in any of these aspects that violate hard-coded, undocumented design choices quickly lead to unforeseen situations such as: unnecessarily slow DAW executions, underutilized resources, sudden runtime errors, straggler processes, meaningless log entries, non-terminating executions, overflows of buffers (memory, disk, log-space), etc. or, in the worst case, undetected faulty results [13]. The execution often stops with an arbitrary, undocumented, low-level error (‘file not found’, ‘core dumped’, ‘timeout’); even meaningful error messages are often difficult to trace back to the broken task as execution happens on multiple nodes and logs are distributed and created at different levels, ranging from OS to resource managers, workflow engine, and task implementations. While some of these problems also occur in other software-related situations, they are aggravated in DAWs due to their heavy reliance on external programs, generally very high resource requirements, long run times, and the complexity of coordinating distributed executions. Accordingly, reusing a DAW on another infrastructure or for input data with differing properties often requires time-consuming adaptations [14,15].

In this work, we propose validity constraints (VCs) as a new primitive for DAW languages that help to improve this situation. A VC is a constraint that specifies a logical condition for a particular state or component of a DAW execution. When a VC evaluates to false (i.e., if the VC is ‘broken’), the DAW engine can issue a defined error message at a defined place. VCs may, for instance, control properties of the input and intermediate data files (e.g., minimal or maximal file sizes), of the runtime environment (e.g., minimal available memory or threads), or of the individual task executions (e.g., maximal execution time). We propose to specify VCs within the DAW specification, i.e., as first-class objects of the DAW program itself. Note that similar ideas have proven extremely useful in other fields (e.g., integrity constraints in databases or contracts in software engineering), but an adaptation to the specific field of workflows is lacking.

In the following, we motivate the idea of VCs for workflows based on a few exemplary user stories from different scientific domains (Section 2) and then first introduce a model for DAWs (Section 3) and then use this model to formally define general validity constraints (Section 4). We present a broad list of different concrete types of VCs (Section 5) and classify these along multiple dimensions, namely the time points when they need checking, the objects they affect, the actions they may trigger, and the infrastructure component that should handle them (Section 5.1). We relate VCs to similar concepts in other fields (Section 6.1), current workflow systems (Section 6.2) and discuss more general works of the scientific workflow community (Section 6.3). Furthermore, we sketch a prototypical implementation of explicit VCs in the state-of-the-art workflow engine Nextflow (Section 7).

Throughout this work, we focus on simple DAWs performing batch processing and leave an extension to data analysis over streams (e.g., [16]) or to DAWs including cycles or conditionals for future work.

2. User stories

We collected a small set of typical problems users from different application domains ran into when using and porting DAWs to another platform. Often, they stumbled over and had to solve validity constraints that were implicit and not explicit.

2.1. Bioinformatics

In bioinformatics research, we often modify or rewrite workflows, which requires developing short workflows performing RNAseq data treatment. We have to check the overall results for their quality and reasonability, but we are also interested in the effects our modifications may cause on different infrastructures performance-wise.

Workflow development—Empty and faulty files. During the development phase, errors can occur, and faulty files may be written. Such a situation does not necessarily interrupt the workflow directly because output files may exist. Workflow engines typically do not assess a task’s success by using the content of the output files. Identifying the wrongly behaving task in a distributed execution environment can become tedious and time-consuming for the user. For example, we once wanted to sort a file in the middle of a workflow but made a syntax error, which caused an empty output file. In this case, it would have been wonderful if we had a language with validity constraints that would help develop workflows and throw an error when an output file is empty, under a certain size, or does not contain specific characters. For such checks, the addition of monitoring tasks is necessary.

Porting workflows to new infrastructures. We recently studied the impact of applying map-reduce on specific bioinformatics tools. While porting a workflow on a heterogeneous distributed infrastructure, we observed a severely reduced workflow runtime. It turned out that only a few nodes could run tasks that should all run in parallel. We found the memory of most of the nodes too small to run these tasks. As a result, we changed the biological model to one with smaller input references and recomputed several experiments. Fortunately, that was possible in this case. But it may not be an option for biologists studying a specific specie, for example. If their reference genome file is too big for the memory of the nodes of a cluster, they would need to set up their workflow on another infrastructure. As such experiments can take a long time (up to 40 h for treating only one sample), a way to know beforehand that the workflow cannot run (with the full degree of parallelism) on this infrastructure can save a lot of time and shared resources. Instead of executing a workflow by checking tasks’ resource demands only late, when the execution reaches them, a basic overall resource check to stop the workflow from the beginning (before it arrives at the task that breaks the workflow) would be preferable.

> *“It would be wonderful if I had a language with validity constraints that would help me develop workflows or port them to a new infrastructure. I would have needed constraints that throw an error when an output file is empty, is under a certain size or does not contain specific characters such as those contained in a specific header. Additionally, some constraints that would stop the workflow from the beginning, before it arrives at the task that requires too much resources, would be very helpful”.*
>
> – Ninon De Mecquenem

The derived user requirements for validity constraints are: file must exist *(R1);* file is not empty or has a minimum size x *(R2);* file has to contain (only) certain characters *(R3);* global pre-check for resource demands *(R4)*.

2.2. Materials science

The novel materials database laboratory (NOMAD) is a database that hosts hundreds of millions of material science simulation results, specifically density functional theory (DFT) simulations [17]. These results can be expensive to generate, sometimes taking several hundred GPU hours to compute [18]. As such, the community realized it is vital to share the results to avoid recomputation and to allow for the creation of large datasets for machine learning and data analysis applications [19,20].

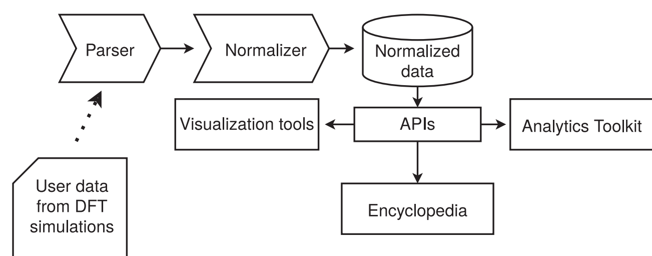


Fig. 1. Overview of the NOMAD upload workflow. Users upload data from a specific DFT simulation code that is then parsed and normalized to make results from different DFT codes comparable.

Users can upload data (input/output files) from different density functional theory simulation programs (e.g., VASP, `exciting` FHI-aims) via the terminal or their browser [21–23]. The workflow to process uploaded data has a clear need for validity constraints. Each simulation program has an associated parser to parse the simulation input/output files that the user uploads [24,25]. A common pain point is that these simulation programs get updated frequently, and the format of the output files change or new data fields are added. The NOMAD developers implemented VCs in the upload DAW to check for the existence of specific file names, extension types, and some properties of the input/output files that the parser expects [21]. For the DFT code `exciting` for example, we expect files named `INFO.out` and `input.xml` files and particular key–value pairs, such as the 'total energy' key and its associated floating point value. This means our DAW first runs a validity constraint at the setup of the upload process to check that the required files exist. If these simulation files do not exist the DAW returns an error message to the user that the upload failed since no parsable files were found. The upload process also triggers a resource availability validity constraint that checks whether the user has used up the amount of storage space allocated to every user of NOMAD.

If, however, the uploaded files satisfy the conditions above, the DAW then runs a resource validity constraint check. The parsing process can take considerable resources depending on the simulation settings and the DAW only executes the parsing once the container orchestration system, in this case Kubernetes, can allocate sufficient computing resources on the server. After parsing raw values from the uploaded files, a routine called the 'normalizer' is applied, which converts parsed values and simulation settings to standard units and standardized terms. For instance, two simulation programs might use different words for the same input parameter and NOMAD needs to standardize this (e.g., two different names for the same DFT functional). The normalizer also implements validity constraints on these parsed values to ensure they are reasonable. For instance, the normalizer checks that a parsed categorical property belongs to a list of expected values or that a floating point value is within a reasonable range (e.g., the band gap of the material must be non-negative). A visualization for the DAW for NOMAD can be seen in Fig. 1.

When we have too many atoms in a unit cell of an input geometry, we observe another weakness of the DAW regarding the crystal structure classification, which is performed in the normalizer step. Large input files with many atoms in the unit cell are common in studies that investigate the effect of impurities on the electronic structure of crystalline materials [26]. Such a situation causes the crystal structure classifier to take a very large amount of computational resources. Currently, we use a timeout validity constraint that stops the classification if the classification takes too long. What might help, however, is to implement a validity constraint that decides whether to skip the crystal structure classification for unit cells where the number of atoms exceeds a certain threshold. This could avoid wasting resources on trying to classify systems that are very likely to trigger the timeout validity constraint during classification. Such a threshold could be determined using a logistic regression model trained on previous uploads and workflow executions. Alternatively, the threshold could be dependent on the resources available for computation. In this case, we envisage VCs that are chained together. Meaning, first we check if resources are somewhat limited and if so, we call a VC that checks if the number of atoms in the unit cell is too large. If this is true, it avoids the crystal structure classification all together.

A further valuable addition to the NOMAD upload DAW would be a check for a metamorphic relation between input settings of the simulation and output results of the simulation to predict data quality [24]. Simulations uploaded to NOMAD usually come from different applications. They could be, for instance, super-high-precision ground state calculations (e.g., using very large basis set sizes) or simulations to find heat and transport properties using cheap calculations (small basis set size), resulting in data of varying precision [27]. Current research has been on using machine learning models to add prediction intervals that act as error bars to the data results parsed by NOMAD [28]. Such annotations would help users better understand how precise and, therefore, how qualified results from NOMAD are for a particular use-case. For example, a formation energy calculated with high-precision settings would have small prediction intervals. It shows the NOMAD end user (e.g., an experimentalist) that this result does not need a recalculation.

Many VCs are already integrated into the NOMAD DAW but we would like to add more as discussed above. A language for clearly presenting the VCs of the DAW would be useful, especially in a graphical form in a visual format. We believe this would help end-users and developers better understand the DAW especially when the upload process fails due to one of these constraints.

Overall, we observe the demand for the following VCs from this user story: file must exist *(R1);* file with certain file extension must exist *(R5);* file content should fulfill certain properties (e.g., contain particular key–value pairs) *(R6);* disk quota check *(R7);* check of resource demands in advance *(R8);* reasonability checks on output data (e.g., value in certain range, positive, non-negative, negative, or from list of given values) *(R9);* tasks finish within a given timeout *(R10);* let workflows decide how to proceed based on the result of VCs (e.g., VC-dependent tasks and chained VCs) *(R11);* data quality checks with metamorphic relations *(R12).*

2.3. Earth observation

In the field of Earth Observation (EO), we typically work with large volumes of satellite images, often over large areas (from federal states to continents, sometimes even global) and across long time series (up to 40 years). Our workflows often consist of a preprocessing step to convert the 'raw' data into a more analysis-ready form. This preparation typically includes the detection of clouds and their shadows, eliminating atmospheric and other radiometric distortions, and often rearranging the data into so-called data cubes for improved data management and efficiency. This processing step usually runs on each individual satellite image separately, allowing image-based parallelism. Subsequently, workflows generally use map/reduce operations on spatial partitions of the data cube, or even sub-partitions that need specific parts of the images. Typically, all available data for some time period is reduced (averaged in the simplest case) to generate gap-free predictive features. The feature vectors at the locations of reference data are extracted, a machine learning model trained with some response variable (e.g., land cover labels or tree height values), and the model applied to the feature images to generate a wall-to-wall prediction of the respective response variable (i.e., generating a map). A validation procedure typically follows.

Different IT resources, such as CPUs, RAM, and I/O bandwidth, typically constrain various components of this generic workflow.

Depending on the particular workflow, the analyzed data, user parameterization, as well as characteristics of the hardware, the limiting factor might be different each time. For example, a workflow that efficiently runs with Landsat data might fail when switching to Sentinel-2 data (more RAM needed) even when executed with the same parameterization on the same system. Another example would be a workflow that efficiently reads data but would quickly become input-limited when switching from an SSD- to an HDD-based platform, or another RAID configuration. In a noteworthy instance, we encountered an extreme worst-case scenario in which processes would sporadically become defunct while unzipping files from tape storage. As a result, our job would come to a complete halt after a certain period. Resolving this issue required a specific modification of our workflow, including the addition of a hard-coded timeout. Moreover, we had to first copy data from tape to warm storage before executing the workflow again.

Consequently, a one-fits-all default parameterization is usually not feasible, and many user parameters may exist that can tweak the behavior of the workflow. For example, the FORCE software [29] includes parameters to fine-tune partition sizes, reduce the number of parallel processes, or to increase the multithreading to multiprocessing ratio when RAM becomes an issue. However, achieving optimal parameterization needs a deep understanding of the workflow, the underlying data, and their effects on system resources. Additionally, a solid understanding of the platform is essential for effective parameterization. Therefore, the presence of validity constraints capable of identifying common patterns of excessive resource usage, such as idle CPUs or high network latency in I/O-limited scenarios, or memory swapping leading to generic 'killed' messages in RAM-limited situations, would significantly aid in transferring workflows from one system to another. Additionally, it would facilitate a smoother onboarding process for new users, reducing the learning curve required.

From this user story, we mainly derive monitoring validity constraints: warn on low CPU usage *(R13);* warn on high network latency *(R14);* warn on memory swapping *(R15)*.

3. Fundamentals

In this section, we formally define a DAW and its execution steps, sketch an abstract infrastructure for executing DAWs in a distributed system, and introduce scheduling as the process of executing a DAW on an infrastructure. Based on these models, we then define two types of general validity constraints (static and dynamic) as new first-class primitives for DAW specification languages, and use them to derive the concept of valid and correct DAW execution.

Our DAW semantic is simple by intention; its purpose is to lay the grounds for the following sections, which will precisely define the connection between elements of a DAW and VCs and the impact that VCs may have on DAW execution. Conceptually, our semantics is similar to Petri-Nets [30] and dataflow languages [31]. Elaborated semantics of real workflow systems have been described elsewhere (e.g., [32,33]); [34] gives a nice overview of different formal models in distributed computation.

3.1. A formal model of DAWs

We define a logical DAW (see below for the distinction to physical DAWs) as follows.

Definition 3.1 (*Logical DAW*)**.** A logical DAW W is a directed acyclic graph

$$W = (T, D, L, \varphi, t_s, t_e) \tag{1}$$

where T is the set of tasks, $D = \{(t', t) \in T^2\}$ is the set of dependencies between pairs of tasks, L is a set of labels, $\varphi : D \to L$ is a function assigning labels to dependencies, $t_s \in T : \nexists (t', t_s) \in D$ is the start task, and $t_e \in T : \nexists (t_e, t') \in D$ is the end task. Intuitively, tasks are the programs to be executed for performing individual analysis steps, while dependencies model the data flow between tasks. The dependencies' label is an abstract representation of the specific data that is exchanged between two tasks. The start task t_s does not depend on any other task and initiates the first steps of the analysis by sending the DAW's input data to its dependent tasks. Similarly, the end task t_e has no dependent tasks, and the labels of its incoming dependencies represent the results of the DAW. Note that having single start and end tasks does not restrict the model significantly as such tasks can easily be introduced as auxiliary, empty additional tasks in front of a group of initial tasks or behind a group of finishing tasks to join them to a single end task. Fig. 2 (upper part) shows a graphical representation of an example DAW consisting of six tasks plus start and end tasks; arcs represent dependencies.

DAWs are executed by running their tasks in an order in which at all times all dependencies are satisfied. To formally define this semantics, we introduce the notation of the state of a DAW and, later, that of valid states.

Definition 3.2 (*State of a DAW*)**.** The state S^W of a DAW W is a function that assigns each task in the set T to one of three possible states:

$$S^W : T \to \{\mathsf{E}, \mathsf{F}, \mathsf{O}, \mathsf{R}\} \tag{2}$$

Here, E means 'executing', F means 'finished', O means 'open', R means 'ready'.

Definition 3.3 (*Valid States*)**.** The state S^W of a DAW W is *valid*, iff the following conditions hold:

(a) $S(t_s) = \mathsf{F}$,
(b) $\forall (t', t) \in D\ \forall t \in T$ with $(t', t) \in D$: if $S(t) = \mathsf{R}$, then $S(t') = \mathsf{F}$,
(c) $\forall (t', t) \in D\ \forall t \in T$: If $\forall t'$ with $(t', t) \in D$: $S(t') = \mathsf{F}$, then $S(t) \in \{\mathsf{R}, \mathsf{E}, \mathsf{F}\}$, and
(d) for all other $t \in T : S(t) = \mathsf{O}$.

The initial state S_0 of a DAW W is the state in which (1) the start task is finished: $S_0(t_s) = \mathsf{F}$, (2) all tasks t' depending on t_s are ready: $S_0(t') = \mathsf{R}$, and (3) all other tasks have state 'open'.

Intuitively, these rules guarantee that: (a) the start task is always in the 'finished' state F; (b) a task t is 'ready' (R) only when all its predecessors ($\forall t \in T$ with $(t', t) \in D$) are 'finished' (F); (c) any task t with all its predecessors t' 'finished' (F) has state 'ready' (R), 'executing' (E), or 'finished' (

3.2. DAW infrastructure and execution semantics

Based on a DAW's state, we next define the semantics of a DAW execution.

Definition 3.4 (*Execution of a DAW*)**.** An execution E of DAW W is a sequence of states $E = \langle S_0, \dots, S_n \rangle$ such that (a) S_0 is the DAW's initial state, (b) all $S_i, i \in \{0, \dots, n\}$, are valid, and (c) for all steps S_i, S_j with $j = i + 1$, it holds that

- If $S_i(t) = \mathsf{F}$, then $S_j(t) = \mathsf{F}$
- If $S_i(t) = \mathsf{R}$, then $S_j(t) \in \{\mathsf{R}, \mathsf{E}\}$
- If $S_i(t) = \mathsf{E}$, then $S_j(t) \in \{\mathsf{E}, \mathsf{F}\}$
- If $S_i(t) = \mathsf{O}$, then $S_j(t) \in \{\mathsf{O}, \mathsf{R}\}$
- There exists at least one $t \in T$ where $S_i(t) \neq S_j(t)$.

We say that an execution E of a DAW W has executed W when $S_n(t_e) = \mathsf{F}$.

Intuitively, the execution of a DAW progresses by iteratively executing tasks that are ready to run. During execution, they are in E; after

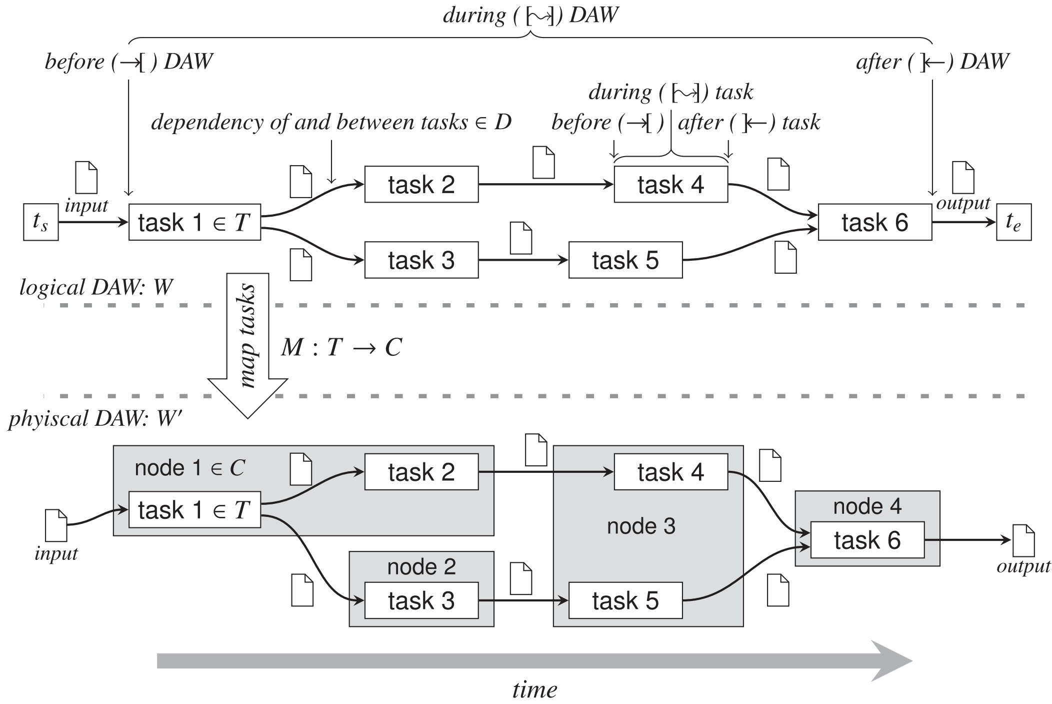


Fig. 2. A logical (upper part) DAW and its physical counterpart (lower part).

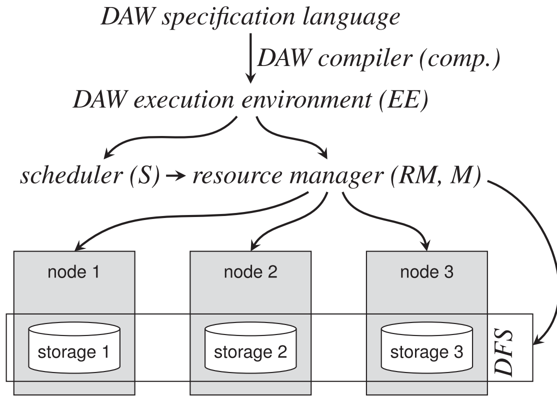


Fig. 3. A simple DAW infrastructure architecture.

execution, their state switches to F; tasks in state O must first proceed to state R before they can be executed. We make no assumptions regarding the order in which tasks that are ready to run at the same state are executed, nor do we assume only one task executes per execution step. But we do require at least one task to change its state between successive DAW states. Note that this change may be purely logical, by switching some task's state from O to R.

Logical DAWs are abstract objects. However, in real life, a DAW execution requires the start of programs representing workflow tasks on particular nodes of the available cluster and the management of the inputs and outputs of these programs. Fig. 3 depicts a light architecture of the components involved in such a DAW execution. It encompasses the DAW specification in a proper DAW language and its compiler (*comp.*), the DAW engine steering the DAW execution (EE), a scheduler performing the task-to-node assignments (S), a resource manager and monitoring system controlling the resource assignment and task execution at the global and local level (RM, M), the individual nodes for executing tasks, and a distributed file system for data exchange between tasks (DFS).[1] Clearly, many other architectures are possible, but for the sake of this work, such an idealized architecture suffices and allows later determining the responsible component to control a particular type of VC. With such an architecture in mind, we can now define a physical DAW.

Definition 3.5 (*Physical DAW*)**.** Given a logical DAW W and a set C of compute nodes interconnected by a network, the physical DAW $W' = (W, M)$ augments W with a function $M : T \rightarrow C$ that maps every task to a compute node.

C is an abstract representation of a compute cluster, whereas M is an assignment (schedule) that maps tasks to nodes; in practice, this assignment is determined by the scheduler when a task's state changes from R to E. The definition of a *DAW execution* E can be naturally extended from logical DAWs to physical DAWs.

Fig. 2 shows an example of the transition of a logical DAW to a physical DAW. In the physical DAW, each logical task is assigned to a node for execution, and each dependency is implemented as communication between nodes.

4. A formal definition of VCs for DAWs

Having introduced logical and physical DAWs and their execution semantics, we can now define validity constraints (VCs) as logical formulas over the components of a DAW infrastructure and of a DAW execution, i.e., tasks, dependencies, executions, and schedules. Different VCs will address various properties of these components (see Section 5).

Definition 4.1 (*Properties*)**.** Let $W = (T, D, L, \varphi, t_s, t_e)$ be a DAW and C be a cluster, i.e., a set of compute nodes. We model arbitrary properties of elements of T, D, and C as property functions:

P_T is a function that assigns properties to tasks from T,

[1] Of course, other means of data exchange, such as in-memory channels, or mounting of remote file systems are possible as well.

P_D is a function that assigns properties to labels of dependencies from D, and
P_C is a function that assigns properties to nodes from C.

We make no assumptions on the specific nature of such properties, such as data type or number of parameters they take. In Section 5, we will give a diverse list of concrete (static or dynamic) properties that we consider particularly useful for DAW management.

4.1. Validity constraints

We discern two types of validity constraints: *Static VCs* address static properties, i.e., properties which always have the same value for a given property of a task/node/label, while *dynamic VCs* address dynamic properties, i.e., properties whose value may change during a DAW execution.

Definition 4.2 (*Static VC*)**.** Let $W = (T, D, L, \varphi, t_s, t_e)$ be a DAW and C be a cluster. Let P_T, P_D, and P_C be their respective property functions. A static validity constraint v is a Boolean formula whose atoms have any of the following forms (with C being an arbitrary constant and $\boxdot \in \{=, <, >, \leq, \geq\}$ and ρ being a given particular property to compare with):

- $P_T(t).\rho \boxdot C$ for any task t from T,
- $P_D(\varphi(d)).\rho \boxdot C$ for the label $\varphi(d)$ of any dependency d from D, and
- $P_C(c).\rho \boxdot C$ for any node c from C,

We call VCs of these three forms *static* because they are independent of a DAW's execution. Intuitively, this implies that they must evaluate to the same value all the time before, during, and after a workflow's execution. An example of a static VC would be the minimum size of main memory that must be available on a node on which a given task is about to be scheduled, or the availability of at least one node with a given minimum memory size within the cluster. The definition also captures conjunctions or disjunctions of atomic constraints; the order in which the individual atoms are checked in an implementation is not determined and leaves room for optimizations [35].

The second class of VCs are *dynamic* VCs, which constrain properties of tasks, dependencies, or nodes that may change during a DAW's execution. For instance, executing a particular task in the middle of a DAW may require the existence (or 'minimal size' or 'a certain format') of a file that is created by previous steps in the very same DAW execution. Introducing such dynamic VCs requires first defining the scope of a step within an execution.

Definition 4.3 (*Scope of Execution Steps*)**.** Let $W = (T, D, L, \varphi, t_s, t_e)$ be a DAW, E an execution of W, and M a schedule for W over a cluster C. Let P_T, P_D, and P_C be their respective property functions. Furthermore, for a step s from E, let $X(s) \subset T$ be the set of tasks of W that are executed in this step, i.e., whose state changes from R to E or from E to F; let $Y^i(s) \subset D$ be the set of dependencies from which tasks in $X(s)$ depend (incoming edges of all tasks executed in this step, each representing an input for a task); let $Y^o(s) \subset D$ be the set of dependencies outgoing from tasks in $X(s)$ (outgoing edges, each representing an output of a task); and let $C(s,t) \in C$ be the node on which M schedules task $t \in X(s)$ for execution. We call the tuple $scope(s) = (X(s), Y^i(s), Y^o(s), C(s,t))$ the scope of step s.

Definition 4.4 (*Dynamic VC*)**.** A dynamic validity constraint V over the scope $scope(s) = (X(s), Y^i(s), Y^o(s), C(s,t))$ of a step s of a valid execution of a DAW W is a Boolean formula whose atoms have of any of the following forms (with C being an arbitrary constant, ρ being a given particular property to compare with, and $\boxdot \in \{=, <, >, \leq, \geq\}$):

- $P_T(s).\rho \boxdot C$ for a property of any task $t \in X(s)$,
- $P_C(s,t).\rho \boxdot C$ for a property of node $c = C(s,t)$,
- $P^i_D(s).\rho \boxdot C$ for a property of a label $\varphi(d)$ with $d \in Y^i(s)$, and
- $P^o_D(s).\rho \boxdot C$ for a property of a label $\varphi(d)$ with $d \in Y^o(s)$.

4.2. Correct DAWs and correct DAW executions

We defined VCs as logical constraints on the components (static) or steps (dynamic) of a DAW that evaluate to either true or false. However, we so far did not describe what consequences the evaluation of a such a constraint should have. Intuitively, VCs are intended as sensors, where an evaluation to 'true' implies that no issue was detected, while an evaluation to 'false' points to a concrete problem. To formally define this intuition, we next introduce the notion of correct DAW setups, where a DAW setup is a combination of a concrete DAW and a concrete cluster on which it should be executed, and correct DAW executions. Recall that we discerned static VCs, whose evaluation always returns the same result for a given combination of DAW and cluster, from dynamic VCs, which are defined over *executions* of DAWs.

Definition 4.5 (*Correct DAW Setup*)**.** Let $W = (T, D, L, \varphi, t_s, t_e)$ be a DAW, C a cluster, i.e., a set of interconnected compute nodes, and V a set of static VCs over P_T, P_D, and P_C. We say that the tuple (W, C) is correct with respect to V when all constraints in V evaluate to true.

We will omit C when it is clear from the context and simply say that W is correct for V.

Definition 4.6 (*Correct DAW Execution*)**.** Let $W = (T, D, L, \varphi, t_s, t_e)$ be a DAW, E an execution of W, M a schedule for W over a cluster C, V^s be a set of static VCs over W and C, and V^d be a set of dynamic VCs over W, E and C. We say that E is correct if and only if:

- (W, C) is a correct setup for V^s, and
- All $v \in V^d$ evaluate to true in all steps $s \in E$.

Accordingly, an execution is not correct whenever either one of the static constraints is hurt or one of the dynamic VCs in at least one step of the execution. We call a step $s \in E$ for which all $v \in V^d$ hold a *correct step*; all other steps are called *erroneous*. Naturally, the first erroneous step is of particular importance, as usually (but not necessarily) DAW execution will stop at this point.

Note that within a step of the execution, a given task's state may change from O to R, from R to E, or from E to F (or may not change at all). This means that a dynamic constraint may affect (1) the start of a task (from R to E), which corresponds to the definition of task pre-conditions; (2) the termination of a task (from E to F), which corresponds to the definition of task post-conditions; or (3) a property of a task while it is executed (within state E), which corresponds to the definition of task runtime-conditions. This distinction has consequences for a practical implementation, because the latter case (3) must be achieved through continuous monitoring of state executions, while the former two cases (1) and (2) can be controlled during state changes, which correspond to defined points in the communication between the scheduler and the execution engine.

Our notion of VCs clearly has limitations in terms of expressiveness, and we can envision several extensions. For instance, we only introduced VCs that affect single steps, single tasks, single dependencies, and single nodes. Thus, we have no notion for expressing constraints that, for instance, ensure (1) that two consecutive tasks in a workflow are scheduled on the same node (because we know of side effects not modeled in the DAW), or (2) that the total size of all intermediate files may not exceed a certain threshold (because there is a quota on available disk space). Both are realistic cases: (1) is a typical requirement emerging when tasks have to be integrated in a DAW that do not

adhere to the fundamental assumption in scientific workflow systems that tasks only communicate through their input/output relationships. Such a demand is often solved by wrapping both tasks into one script, which, however, blurs dependencies and removes degrees-of-freedom for the scheduler. (2) is a common case in clusters shared between different groups to ensure a fair share of resources. In current systems, such quotas are controlled by the file system which would block write requests beyond this limit, which in turn very likely would cause the workflow engine to crash or the involved tasks to be aborted after their execution timeout, which may complicate finding the root cause.

Extending our model of VCs to cover such cases would be worthwhile yet non-trivial. Regarding (1), our model of VCs would need to be extended by constructs that express conditions on pairs of tasks; a further generalization would allow selecting arbitrary subsets of tasks, possibly by some property (such as all tasks requiring a GPU), which, in turn, would need a model for annotating tasks and nodes. A more restrictive extension would allow only constraints on pairs of successive tasks, which would suffice for the example given. Similarly, a VC for example (2) would need to add constraints on groups of files. Allowing such extended forms of VCs would also impact their implementation. In their present form, VCs can be checked at clearly defined positions during a workflow execution; in contrast, a VC like (1) with arbitrary groups of tasks would require checking whenever a task of the group is started or finished, and a VC like (2) would require checking whenever an output file is generated or written to. We leave such extensions for future work.

5. Concrete validity constraints for DAWs

After having defined DAWs, their components, VCs, and the formal relationships between DAWs and VCs in an abstract manner, we shall now introduce a broad collection of different concrete VCs. We do not aim for completeness but for a representative set that illustrates the spectrum of functionalities that can be covered when using VCs as first-class primitives for DAW languages. Naturally, one can envision further constraints up to arbitrary user-defined VCs, provided a proper specification language for them is defined. Note that none of the VCs we discuss is completely new; instead, many of them can be found either implicitly or explicitly in other research fields, such as integrity constraints in databases or pre-/post conditions in programming languages; we shall discuss these related lines of research in Section 6.

We shall present VCs in three steps. We shall first list them grouped by the component of a DAW system they address, namely setup, task, file, or user-defined accompanied by an intuitive explanation and a classification into 'static' or 'dynamic'. In Section 5.1, we distinguish several properties of VCs to enable a more fine-grained distinction. These properties will allow to systematically characterize VCs in Section 5.2.

Based on related ideas in other fields, own experience in DAW development, and the user stories in Section 2, we consider the following VCs as particularly important. We group them according to the part of a DAW/infrastructure they primarily affect. Note that not all of them have the same level of abstractions; in some cases, we rather describe a type of VC than a concrete VC. For instance, we introduce a general VC for file properties instead of one distinct VC for every such property.

Setup-related VCs

This set of VCs are related to the particular combination of a DAW and the cluster it should be executed on. Two of them are intended to be controlled before the DAW execution starts and are thus static. The third is inherently dynamic.

static resource availability: The nodes within a cluster must fulfill certain requirements in terms of available resources, such as minimal main memory, minimal number of allocated CPU hours, or availability of a GPU of a certain type. This VC could be defined with two different semantics: In *at-least one node*, at least one node of the cluster must fulfill the constraints; in *all nodes*, all nodes must do so. This VC addresses requirements *R4, R7,* and *R8* of Section 2.

file must exist: File must exist and must be accessible. Thus may, for instance, affect certain reference or metadata files, but can also be used to ensure availability of input files of the DAW. This VC could also be defined either in *at-least one node* or *all nodes* semantics; however, the latter is more common. This VC addresses requirement *R1*.

infrastructure health: A node responds to requests from the DAW engine prior or during a DAW execution. Such constraints are often implemented with the help of a heartbeat-style infrastructure.

Task-related VCs

Task-related VCs describe properties of a concrete task of a DAW. Many of them can be defined either statically or dynamically.

executable must exist: During execution, any concrete task must be scheduled on some node in the cluster. The program executing this task must be available on this node. Can be defined statically, which requires that all nodes in the cluster maintain executables of all tasks in the DAW, or dynamically, which allows for temporary installation (and subsequent deletion) of executables of tasks as part of their scheduling.

dynamic resource availability: Before starting a task on a given node, certain requirements in terms of available resources must be fulfilled, such as minimal main memory, minimal number of allocated CPU hours, or availability of a GPU of a certain type. During task execution certain thresholds of resource usage have to be met. This VC addresses requirements *R7, R8,* and *R13–15*.

configuration parameters: Parameters for execution of a task within a DAW must be valid, e.g., have a value within a certain range or of a certain format. Is typically defined dynamically as many arguments of tasks are created only at runtime, such as the names of input/output files.

license valid: Some tasks might require a valid license to start. Can be defined statically (test for general availability of a valid license for all tasks in a DAW) or dynamically (test for concrete availability of a valid license as part of task scheduling). The latter is important then the number of possible concurrently running tasks is constrained by a volume contract.

metamorphic relations: The relation of input and output of a task can be characterized by a reversible function. After executing a task, the concrete pair of input/output must have this relationship. This VC addresses requirement *R12*.

tasks end within limits: The runtime of a particular task can be constrained by a VC on its maximal runtime. Such a constraint can help to identify stragglers. This VC addresses requirement *R10*.

task ends correctly: The execution of a particular task must end with a predefined state or output message.

File-related VCs

File-related VCs control the management of files within the infrastructure. Using our definitions from Section 3, this also encompasses dependencies and hence data exchange between tasks.

file must exist: File must exist and must be accessible before starting a task on a node. This VC addresses requirement *R1*.

file properties: A file must fulfill certain criteria, such as file size, format, checksum over content, or creation time. Can be defined statically (properties of metadata files) or dynamically (properties of files generated during DAW execution). This VC addresses requirements *R2, R3, R5, R6,* and *R9*.

folder exists: Certain folders must exist and must be readable before/after execution of a DAW/task.

User-defined VCs

In practical applications, DAW developers often find very specific cases of validity constraints, which cannot be captured in a pre-defined catalog as the one just presented. For such cases, systems should also foresee user-defined validity constraints, which implement arbitrary custom code. These must be executed before or after a workflow is run or a task is started, depending on the specific definition. Naturally, user-defined VCs cannot be categorized well as their custom code can perform arbitrary computation and access arbitrary data (within the execution environment of the controlled object). Therefore, we will not further describe user-defined VCs in the following sections. Note that our prototypical implementation presented in 7 also allows and uses user-defined VCs. Such VCs may help addressing elaborate variants of requirements *R3, R5, R6, R9, R11,* as well as *R13–15*.

5.1. Properties of VCs

We so-far classified VCs only broadly into two classes based on the point in time when they can be checked in principle. However, there are many more dimensions by which VCs can be characterized. For instance, violations of VCs can have different levels of severity; while some must result in an immediate stop of the DAW execution, such as in the case when a task in the DAW requires an amount of main memory that none of the nodes of a cluster can provide, others might be interpreted rather as a warning, such as an improbable yet not impossible file size. Some VCs must be checked before a task starts, such as the available resources on the node it is scheduled on, some after a task ends, such as its result status, and a third class of VCs requires continuous control during task execution, for instance to ensure termination within a runtime limit.

Table 1 provides six different properties (or dimensions) by which VCs can be characterized. These dimensions are mostly independent of each other and all have their own importance. For example, knowing whether a constraint is ‘hard’ or ‘soft’ is equivalent to knowing whether it expresses a mandatory requirement or not. The ‘affected object’ informs how to track the constraint and what might be affected if it is violated.

Such a more fine-grained classification for VCs enables differentiating techniques and therefore enables a common shared understanding and objective discussion about VCs. Newly found VCs can be contrasted and grouped with other validity constraints using a given classification. Classifications can also help identifying new VCs, by looking for a VC that fulfills a certain combination of properties.

Table 1
Dimensions by which VCs can be characterized.

Dimension	Description and possible values
Severity	Describes whether a VC must be fulfilled or not; non-mandatory VCs implement plausibility checks. Possible values: *hard* implies immediate stop of DAW execution; *soft:* issues a warning, for instance in the DAW log.
Affected object	Describes the type of object addressed by a VC. This dimension was used to group VC in the previous text. Possible values: *setup*, *task*, and *file*.
Type	Describes the type of a VC. Possible values: *static*; *dynamic*. See also Section 4.1.
Time of check	Describes the point in time when a VC should be checked. Possible values: *before,* →[*:* check before task starts on a given node, when state changes from R to E; *after,*]←*:* check after a task has finished, when state changes from E to F; *during,* [⇝]*:* check periodically during task execution, i.e., while in state E.
Component	Describes the component in the DAW architecture (see Section 3.2 and Fig. 3) which is responsible for controlling a VC. Possible values: *execution engine* EE; *scheduler* S; *resource manager* RM; *monitoring* M.
Recoverable	Describes whether the DAW system can try to recover from the error automatically. Possible values: *yes* (+), *no* (−), *maybe* (±).

5.2. Formal characterization of VCs

In this section, we introduce a classification for Validity Constraints for DAWs. First, we will explain why a classification is helpful in this context. Then, we present the properties alias dimensions we deem relevant to classify VCs and then classify the introduced VCs in Section 5 accordingly.

Using the properties and dimensions of Table 1, we classify the selected VCs from Section 5 in Table 2 and observe the following general trends and traits for the selected VCs. Most VCs are either hard constraints or can be both hard and soft constraints. This characteristic is most likely because there is less value in a VC that never indicates an error. There are two big groups regarding the time a VC is checked: many VCs are checkable either ‘before’ or ‘during’ the execution; few VCs are checkable ‘after’ the execution. This deviation is most likely caused by preconditions and invariants being more common than means to check postconditions. The most predominant component for checking VCs is the EE, which is tightly coupled to almost all of the dynamic VCs, as the EE is inherent to the execution. The time of check also correlates with the discreteness of the checks. If a VCs’ time of check is ‘before’ or ‘after’, it is usually ‘discrete’. If the time of check is ‘during’, the VC is usually ‘continuous’. As we do not have enough examples of ‘triggered’ constraints, we cannot point out similar correlations for those. Many VCs are not limited to workflows but are also applicable in related fields, i.e., they represent more general constraints. Many constraint violations are ‘recoverable’ as violating them can be caused by spurious problems.

6. Related work

In the following, we first look at implicitly and explicitly defined concepts similar to VCs one can find in other research fields. This review also was an important source of input for Section 5. In Section 6.2, we provide a survey of validity checking mechanisms in selected current workflow languages or systems, namely the Common Workflow Language (CWL), Nextflow, Snakemake, Airflow, Spark, and Flink. Finally, Section 6.3 discusses prior (now essentially historical) work in VC-related concepts in scientific workflow research.

6.1. VCs in other fields of research

First, we look at VCs in other fields of research.

Database management systems. Relational databases use tables with attributes and values to thore their data, allowing queries through declarative languages like SQL [36]. They enforce integrity constraints (ICs) [37], which ensure specific properties of attribute values. These can be constraints on individual values (e.g., value-range constraints), constraints on all values of an attribute (e.g., unique constraint), and constraints relating values across attributes (e.g., foreign key constraint). In addition, user-defined constraints can be programmed using triggers for specific actions, such as insertion or deletion of a tuple.

Table 2
Validity constraints for workflows.

Constraint	h(ard)/s(oft)/b(oth)	Affected object	Time of check	Responsible component	s(tatic)/d(ynamic)/p(osthoc)	Recoverable?
Setup-related:						
Static res. availability	h	$P_C(c)$ or $P_T(s)$	→[[⇝]	S,M	d	+
File must exist	h	$P^i_D(s)$, $P^o_D(s)$	→[ ]←	EE	d/p	±
Infrastructure health	b	$P_C(c)$, $P_C(s,t)$	[⇝]	M	d	+
Task-related:						
Executable must exist	h	$P_T(s)$	→[	EE	d	−
Dynamic res. availability	h	$P_C(c)$ or $P_T(s)$	→[[⇝]	S,M	d	+
Configuration parameters	b	$P_T(t)$	→[	EE	d	−
License valid	h	$P_T(t)$	→[	EE	d	±
Metamorphic relations	b	$P_T(t)$	]←	EE	d/p	±
Tasks end within limits	h	$P_T(s)$	[⇝]	S,EE	d/p	±
Task ends correctly	b	$P_T(s)$	]←	EE	d	±
File-related:						
File properties	b	$P^i_D(s)$, $P^o_D(s)$	→[ ]←	EE,RM	d	±
File must exist	h	$P^i_D(s)$, $P^o_D(s)$	→[ ]←	EE	d/p	±
Folder exists	h	$P^o_D(s)$	→[	EE	d	±

ICs in databases resemble validity constraints (VCs) in workflows, but there are key differences. ICs are defined over a persistent database and are enforced on every data change, ensuring consistency. In contrast, VCs for workflows operate in a transient process and must be managed alongside workflow execution. Some VCs prevent inconsistent states (pre-conditions), while others respond after inconsistencies occur (post-conditions). Database systems are monolithic, incorporating IC control, whereas workflow systems involve multiple independent components, making VC control more challenging (see Section 7).

Model checking. Model checking is a technique for automatic formal verification of finite state systems. The model checking process can be divided into three main tasks [38]:

Modeling: Convert a design (software, hardware, DAW) into a formalism accepted by a model-checking tool.

Specification: State the properties a design must satisfy (i.e., some logical formalism, such as modal or temporal logics).

Verification: Check if the model satisfies the specification (ideally completely automatic).

Especially the second task (specification) and the first task (modeling) are related to VCs. During modeling, the DAW is translated into a Kripke transition system [38, Ch. 3], an automaton with states (tasks) and transitions. Each valid path in the Kripke transition represents a valid execution path in the DAW, thus ensuring that the necessary task execution order is respected. Temporal logic [38, Ch. 2] is used, for example, to define VCs locally or globally in the specification. In other words, the constraints can be on the state (task) level, which can indicate that there exists a task along the path that fulfills a given condition or constraint. Furthermore, constraints can also be defined on the path-level, meaning that there exists a path generated from a state that holds true for a given condition [39].

Business Process Management (BPM). BPM studies workflows in business-related areas to improve business process performance. There are different relevant perspectives to consider. The *control-flow perspective* models the ordering of activities and is often the backbone of BPM models. Organizational units, roles, authorizations, IT systems, and equipment are summarized and defined in the *resource perspective*. Furthermore, the *data or artifact perspective* deals with modeling decisions, data creation, forms, etc. The *time perspective* addresses task durations but also takes fixed task deadlines into account. Lastly, the *function perspective* describes activities and related applications [40]. Process models can use conditional events to define business rules. A conditional event allows a process instance to start or progress only when the corresponding business rule evaluates to true. When handling exceptions in BPM, validity constraints can be internal (caused inside the task) or external (caused by an external event) exceptions. Another constraint is the activity timeout, where an activity exceeds the predefined time for execution [41].

Software engineering and programming languages. Software engineering involves designing, implementing, and maintaining software systems. Data types play a crucial role by defining how different components can interact. Using inappropriate data types can lead to unspecified and likely invalid behavior, so enforcing type constraints is vital for software validity.

Most programming languages have type checking. Dynamically typed languages like Python do this at runtime, while statically typed ones like Java do it at compile time. If a type constraint is violated, an error message is returned.

Assertions and exceptions in programming languages check user-defined validity constraints at runtime. They are used for tasks like verifying the existence of a file or specifying correct behavior in software tests. Bertrand Meyer introduced Design-by-Contract in the Eiffel language [42], a methodology for designing reliable systems using assertions, preconditions, postconditions, and class invariants having its roots in Hoare Logic [43] for proving program correctness.

Result checking [44], involves a separate program dedicated to verifying the correctness of results. This differs from software testing in that the checker must meet strict reliability and runtime requirements.

The Rust programming language[2] [45] aims for safety by design, particularly for concurrent programs. It ensures memory safety using a borrow checker that validates memory references and controls access through its ownership system, preventing issues with multiple threads accessing the same variable.

Service composition and interface constraints. The topic of automatic service composition is also the main focus of the book by Tan and Zhou [46]. It discusses, for example, the verification of service-based workflows, quality-of-service (QoS) aspects, deadlock detection, and dead path elimination. Based on interface descriptions they verify the automatic composability of workflows. As a foundation to analyze properties (e.g., deadlock detection) of DAWs Petri nets [30], π-calculus [47], process algebra [48], or automata (linear temporal logic) [49] can be used. When services or tasks are not directly composable, one can look for mediator tasks that make interfaces compatible. In terms of validity constraints, the work mainly focuses on the validity of interfaces and data formats between tasks, which may be overcome with mediators.

Machine Learning Operations (MLOps). MLOps involves the complete process of integrating machine learning models and pipelines into applications. For example, a video streaming service might use a model to suggest content or show relevant ads. The model integration involves reoccurring tasks, such as the curation, filtering, and preprocessing of datasets, plus the design, training, and validation of the models [50].

Ensuring model validity in MLOps focuses on robustness and prediction quality. While it is challenging to ensure correctness throughout the entire pipeline, checks can be made at input and output stages. This includes cross-checking output quality, detecting changes in input data distributions, and verifying infrastructure requirements.

[2] https://www.rust-lang.org/.

Machine learning pipelines are often designed for various infrastructures and complex technology stacks. Although platforms like TFX [50], MLFlow [51], or Kubeflow [52] support MLOps, there is no widely accepted standard to address fully automated validity checking. This is crucial due to the continuous changes in pipelines driven by real-world data, regulations, fairness, and safety concerns. Each change has the potential to introduce errors to the pipeline. Thus, the need for standardized quality assurance in MLOps is evident.

6.2. VCs in current workflow systems

After this overview of validity constraints of different fields, we next describe the state-of-the-art in validity constraint definition and checking in actual systems. To this end, we look at a selection of current popular state-of-the-art workflow systems and examine if and how they support the application of validity constraints.

Common Workflow Language (CWL). The CWL is an open standard that facilitates the description of command-line tool execution and workflow creation. It is still under active development [53]. The ways to define validity constraints are currently limited but subject to extension. So far, CWL supports a dynamic definition of resource requirements enabling the optimization of task scheduling and resource usage without manual intervention. Additionally, it allows the specification of software requirements. Both the resource and software requirements are expressed as hints. Workflow engines may consider or ignore these annotations as CWL is merely a workflow language and standard but does not provide a full-fledged execution engine other than a simple proof-of-concept runner. For better validation of workflow connections, it is recommended to use file format identifiers [54]. An extension currently under discussion is the addition of input value restrictions.[3] Our proposed VCs are more general and conceptually go beyond the basic sanity checks that CWL currently offers.

Nextflow. Nextflow is a workflow system that provides its own domain-specific language to compose user-provided tasks into workflows [55]. Although it is mainly used in the bioinformatics domain, Nextflow can be used to build workflows in any domain. Recently, the Nextflow developer team introduced their new language 'DSL2' to build Nextflow workflows. They point out that the next focus is to take advantage of the improved modularization capabilities of DSL2 to support the testing and validation of process modules. We are not aware of any built-in functionality to define or check validity constraints of the workflows currently, though. Note that in Section 7 we will describe a prototype implementation of VC for Nextflow.

Snakemake. Snakemake is a workflow management system that uses a Python-based language to define and execute workflows. Each rule in Snakemake specifies input and output files, along with any parameters or commands needed to produce the output from the input. The rules can be chained together to form a directed acyclic graph that represents the dependencies between the rules [56]. While Snakemake ensures that each rule is well-defined and the workflow is reproducible, it does not, as far as we know, provide a formal mechanism for specifying validity constraints or checking the correctness of the workflow at runtime. Although, Snakemake supports a dry-run with the command line option '-n' that can be used to check whether the workflow is defined properly and can also provide a rough estimate of the required computational time necessary to execute it. Furthermore, Snakemake checks for the existence of a task's defined output files after its execution. For further checks, such as checking for them to be non-empty, users are advised to implement that by shell commands manually to provoke a non-zero exit status of the task.[4] With VCs as an explicit concept, we can keep the main business logic separate from sanity checks and quality assurance. In a production environment, were everything runs smoothly without infrastructure or other changes, one then can consider to skip the actual checking of some VCs for performance reasons.

Apache Airflow. Apache Airflow is a workflow management system created in 2014 by Airbnb [57]. Workflows in Airflow are created using the Python API. Airflow does not explicitly provide functionality targeted at checking the validity of workflows. Instead, they provide a best practices section in their documentation with a description of testing of airflow workflows. In this description, the authors suggest manually inserting customized checks into the workflow to ensure results are as expected. However, such a check is simply another user-defined task inside the workflow, and there are no specific airflow constructs to help build such checks or to react when checks fails. We discussed benefits of having VCs as a separate concept at the end of the previous paragraph and throughout the paper.

Apache Spark. Apache Spark, which was started in 2009 at UC Berkeley, is a workflow engine for large-scale data analysis [58]. Spark workflows are defined via APIs in Java, Scala, Python, or R. Apache Spark does not seem to support validity constraints for their workflows. Therefore, users need to come up with their own validation schemes.

Apache Flink. Apache Flink is a data analytics engine unifying batch and stream processing [59]. Akin to Spark, Apache Flink workflows are created using Java, Scala, or Python APIs. In a document for the nightly build of Apache v1.15, the Apache Flink team introduces a new non-stable minimum viable product named "Fine-Grained Resource Management". This new feature will allow workflow developers to specify the resource requirements manually for each task. While this feature's primary objective is to improve resource utilization, this may provide the possibility for resource-based validity constraints. Aside from that, Flink offers extensive support for local testing and validating workflows with constructs such as test harnesses and mini clusters.

In summary, the concept of VCs seems not yet to be well established across the different workflow management systems. Nevertheless, the challenges of portability, productivity, and performance of workflows get increasing attention also in related fields such as high-performance computing [60].

6.3. Previous work on VCs for scientific workflows

Scientific Workflows are DAWs in the scientific data analysis domain (Section 1). Typically, they build on a Scientific Workflow Management System, which encompass workflow languages, execution engines, a form of resource management, and a form of data exchange; the later two components are often delegated to infrastructure components like shared file systems or resource managers. Over the years, many such systems were developed with differing features and capabilities [61]. Validity constraints—although they are a vital ingredient for portability, adaptability, and dependability as discussed in Section 1—often remain implicit and unchecked in these systems [3]. Some research addresses only very specialized aspects, such as Rynge et al. who focus solely on detecting low-level data corruption (as hard VCs), for instance, after caused by network or hardware errors [62]. In the following, we discuss some prominent systems or perspectives from the viewpoint of validity constraints.

Semantic workflows. Semantic workflows denotes a class of workflow languages that build on an elaborated, often domain-specific type system or ontology [63]. With this ontology, data that is to be exchanged between tasks are assigned a specialized type (such as "genomic reads from machine X" instead of the basic "set of strings"), tasks are assigned a type (such as "read mapper for genomic reads"), and the IO channels of tasks are assigned types. Types are arranged in a specialization

[3] https://github.com/common-workflow-language/common-workflow-language/issues/764.

[4] https://snakemake.readthedocs.io/en/v7.25.0/ → FAQ.

hierarchy which allows inference regarding type compatibility or workflow planning [64]. For instance, Lamprecht introduced a workflow language that allows for the definition of semantic constraints, leading to methodologies for model-guarded and model-driven workflow design [65]. Another example is the semantics-based 'Wings' approach to workflow development and workflow planning [66].

Types, i.e., semantically defined concepts, combined with compatibility checking are a form of validity constraints. The are usually defined statically and can be checked before workflow execution based on annotation of the workflow components. They operate on a different level than the VCs we defined. However, such systems are (yet) rarely used in practice because they require all data files and all tasks to be used in a workflow to be annotated with concepts from a consistent ontology. In quickly changing fields like scientific research where DAWs are often explorative, this requirement makes development cumbersome and inflexible. It also requires significant effort in community-driven ontology design and maintenance [67].

AWDL based workflows and data constraints. Qin and Fahringer [68] use the abstract workflow description language (AWDL), an XML-based language expressing workflows. AWDL allows describing the directed acyclic graph (DAG) of tasks with their conditions and execution properties (parallel, sequential, alternative paths), etc. Further, it allows specifying constraints for the runtime environment, optimization, and execution steering. The approach follows a UML-based workflow modeling and modularization. The specification of data representations and activity types with ontologies aims for automatic semantic workflow composition and automatic data conversion.

AWDL supports simple properties and constraints per data port and task, such as read-only or read-write data, expected output size, memory usage, required CPU architecture, etc. It also supports constraints on the data distribution like "a task only needs the first index", "a task can work on single items", "a task needs a window of x items", etc. The typed data sources and sinks help the automatic composability of workflows and necessary data conversion tasks.

Temporal constraints. Liu, Yang, and Chen discuss temporal constraints in scientific workflow systems [69]. They argue that fixed time constraints are often too strict, and their violation not necessarily indicates a failing (or otherwise wrong) workflow execution. Instead, they introduce probabilistic temporal constraints, e.g., 90% of tasks of class 'A' finish within 60 min. They distinguish the components of *setting* temporal constraints, *monitoring* temporal consistency, and *handling* temporal violations. Checkpoints can be used for re-execution and temporal checks. To overcome constraint violations they distinguish *statistically recoverable* temporal violations and *statistically non-recoverable* temporal violations. The former can be handled by doing nothing, or rescheduling, and the latter by adding resources, stopping and restarting tasks or workflows, or workflow restructuring.

Provenance and provenance validity constraints.

The *provenance* of information produced by executing a workflow on some input dataset describes a graph of data derivations, from the inputs to the outputs, through each of the intermediate tasks, and potentially including accountability metadata, i.e., who has been responsible for the data and for the workflow specification.

It has been suggested [70] that provenance graphs can help to check reproducibility as well as to validate the workflow's correctness and its performance, though practical tools to achieve this are still lacking. When provenance is encoded using a standardized data model, such as the W3C's PROV,[5] it may be possible to express simple validity constraints on one workflow, for instance to assert that only inputs with specific attribution can be accepted. In the context of reproducibility, more complex constraints involving two or more workflow executions can also potentially be defined, for example to express the acceptable differences between the outputs obtained from two runs.

The PROV data model is designed to facilitate interoperability of provenance information across different data providers and consumers. For this, PROV itself defines a formal system of *validity constraints*, denoted PROV-CONSTRAINTS.[6] The constraints define structural and semantic properties of the graph, asserting for instance that "a data element cannot be used before it has been produced", but also to enable inferences, for example "if a data element is generated by two activities" (for instance, two workflow tasks) "A and B, then A and B must refer to the same activity" (uniqueness of data generation). This is important in the context of provenance interoperability, to ensure that semantic correctness of a provenance document received from a third party "does not tell an impossible story about the data". Constraints have also been used to define 'canonical forms' for PROV graphs [71]. Note that here the constraints are all pre-defined, as opposed to user-defined, and as such they differ in their purpose and practical usage from the type of VCs considered in this paper.

7. Implementing VCs as contracts in Nextflow

We implemented a prototype for a subset of the types of VCs defined in Section 5 in the popular workflow system Nextflow to validate our conceptual model in practice. Details of the implementation and its evaluation are available in [72,73]. In brief, we added two new directives called '`require`' and '`promise`' into the Nextflow specification language, which allow us to insert code for dynamic, task- or file-related validity constraints specifying pre- or postconditions (but not runtime conditions) into task definitions. By incorporating VCs into the workflow definition language, we can leverage the existing language tools available in Nextflow, such as Groovy. This approach avoids a separate VC language with its own syntax and interpretation infrastructure, and VC writers do not have to learn another specification language [74]. Although VCs are written along the main business logic of a workflow, the code for VCs should not perform any essential tasks for the workflow, i.e., the workflow should run properly without the VCs being executed. The two newly added primitives are part of an extension to the DAW model borrowing the concept of contracts from software engineering, described in Section 6.1.

This contract-based approach allows adding a contract to each task in a workflow. Such contracts manifest as sets of requirements and promises checked immediately before ($\rightarrow\!|$) and after ($|\!\leftarrow$) the task execution to ensure that the task runs in the appropriate environment and produces valid results. We added the primitives '`require`' and '`promise`' to Nextflow's workflow definition language, so that code for these contracts can be incorporated into the task definition. These contracts are then executed alongside the tasks on the cluster as arbitrary bash scripts, thanks to Nextflow's nature of compiling each task into a bash script. To facilitate the creation of these contracts, we introduced auxiliary constructs with an internal domain-specific language (DSL) [75] to Nextflow's workflow definition language. Fig. 4 exemplary shows how to define VCs in the form of a contract for a Nextflow process.

When Nextflow generates the command bash script for a process, it now places the code from the require block before the process code and the code from the promise block after the process code. The resulting program can be sent for execution on the cluster nodes. In this specific example, the process requires that all FASTA[7] files should have lines that start with certain characters, such as $\{>, A, C, T, G, U, N, ;\}$. This is a simple check to verify the file format before data processing and makes the workflow more robust for better portability and adaptability. If the preceeding step of data generation is altered and then may use a

[5] https://www.w3.org/TR/prov-dm/.

[6] https://www.w3.org/TR/prov-constraints/.

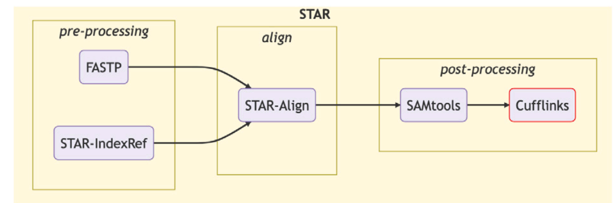
[7] https://blast.ncbi.nlm.nih.gov/doc/blast-topics/.

```
process x {
  input:
  [...]

  require([
    // for all FASTA files (extension .fa)
    FOR_ALL("f", ITER("*.fa"),
    { f -> IF_THEN(
        // check lines' first char
        COND("grep -Ev '^[>ACTGUN;]' $f"),
        // exit if first char is not one of ">ACTGUN;"
        "exit 1")
    }
    )
  ])

  promise([
    // auxiliary function checking stderr
    COMMAND_LOGGED_NO_ERROR(),
    // auxiliary function checking inputs
    INPUTS_NOT_CHANGED()
  ])

  """
  DATA PROCESSING CODE
  """
}
```

Fig. 4. Listing of a Nextflow process with '`require`' and '`promise`' primitives to define VCs.

binary encoding or compression for the data, for example, the VC safeguards the task and user from weird errors or unnoticed wrong analysis results. After processing the data, the task ensures that the process execution command did not encounter any errors and that the input files remain unmodified. These contracts are categorized as *dynamic validity constraints* because they are code that runs alongside the task they are defined in. In terms of categorizing VCs as described in Section 5, these implemented contracts belong to the task-related VCs. The task contracts can dynamically check node- and file-level properties, such as verifying that the current node has sufficient resources. However, they cannot check properties for all nodes. So, they cover the requirements *R1–3, R5–9,* and *R12* of Section 2. The remaining open requirements belong either to global checks or continuous monitoring aspects, which fit not well into the contract-based, task-focused approach but need to be implemented with different means on another level.

We performed a case study in collaboration with domain scientists from the life sciences involving real-world DAWs from the bioinformatics domain, where scientists encountered several issues during development that caused delays which sometimes lasted several days [72,73]. We enhanced these workflows with contracts to test the effectiveness and comprehensiveness of our contract-based approach to implementing validity constraints. This allowed us to identify common problems that arise during their execution and demonstrated that the specific notifications provided by broken contracts aid in debugging the DAWs. Our investigation focused on three main areas: (1) the impact of runtime overhead on each task, (2) the amount of computation time that could be saved by aborting the DAW early, and (3) how contracts enhance issue localization and explanation. Our practical use cases confirmed that the specification even of simple contracts are very effective in supporting the identification of issues in real-world DAWs, that they can save substantial compute time due to early aborts, and that the runtime overhead often is negligible, depending on the type of checks performed. The runtime is illustrated by Figs. 5 and 6 in two exemplary workflows from bioinformatics we studied [72,73]. The contract with the highest absolute costs resembles the one outlined in the listing above, which checks the input files to contain certain characters for the FASTP task. Nevertheless, it needs only about 3.3% of the actual task runtime but may save numerous hours of problem investigation by domain scientists based on the experience we gained in the case study.



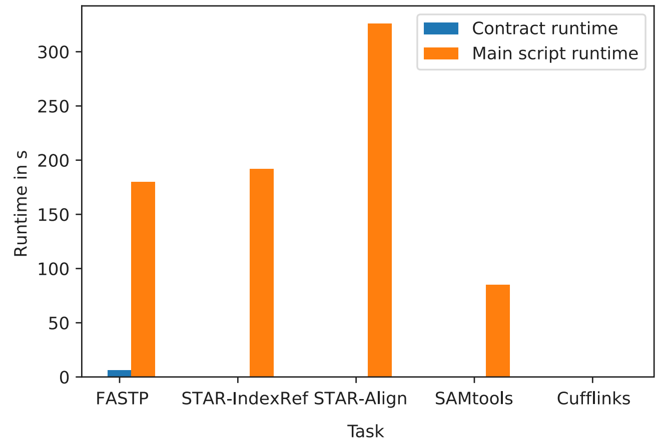
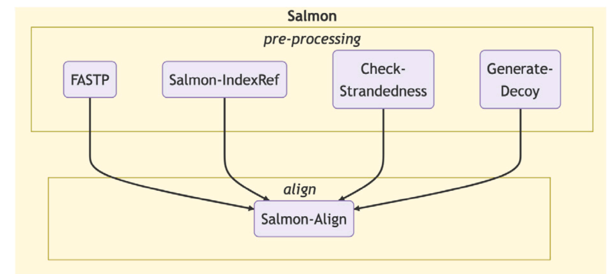


Fig. 5. Exemplary STAR workflow from bioinformatics with task and contract execution times. The Cufflinks task actually failed.



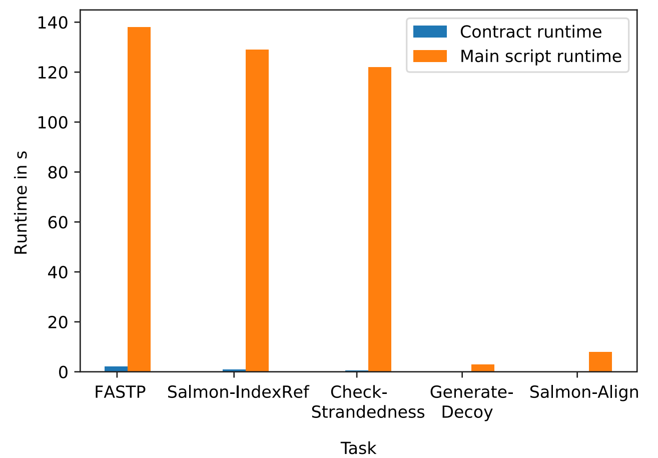


Fig. 6. Exemplary Salmon workflow from bioinformatics with task and contract execution times.

8. Validity constraints specification approach

Throughout this article, we have argued for the role of VCs as a means of ensuring the portability and sustainability of a data analysis workflow across different environments and contexts, ultimately saving time for users and developers who (re)use the workflow. However, an important question arises: How can VCs be specified? Typically, VCs

are established during the final stages of workflow development, once the workflow has been designed, executed and is ready for sharing. It is the responsibility of the workflow developer to specify and incorporate VCs before sharing and publishing the workflow. However, VCs' specification can be laborious and time consuming. Not only does the workflow developer need to consider the specific data and processing requirements within their own execution environment, but they also need to anticipate potential issues that may arise from variations in datasets, formats, parameters and environments. While this may seem like extra work for the workflow developer, it is crucial to ensuring the long-term viability of workflows and their reuse by others. Some constraints can be directly stipulated by the developer, such as required input and output files. Other types of VCs, e.g., setup- and task-related, however, can be tricky to specify without some investigative effort from the developer. In this section, we discuss a few sources of information that can be exploited to assist workflow developers in specifying (at least in part) VCs.

Debugging. Workflows often require multiple debugging iterations before achieving successful execution. During this process, workflow developers execute the workflow using specific inputs, encounter failures, and then proceed to modify the workflow, input data, or environment parameters until a successful execution is achieved. Failed executions of workflows can serve as valuable sources for specifying VCs. If the workflow developers thoroughly document the reasons a workflow execution was deemed to fail, e.g., excessively quick or long task execution, empty output file, as well as identifying the specific causes, then such documentation can be exploited to harvest VCs. In particular, debugging is likely to point-out file-related causes such as missing input file or incorrect input file format, as well as some task-related causes, e.g., the existence of a given executable on a given node.

Testing. While debugging can be effective in helping specify file-based VCs, it is less effective for setup- and task-related constraints. During debugging, the developer focuses on successfully executing the workflow with specific input files of interest. Just like software products, workflow debugging is complemented with testing to ensure that the workflow behaves as expected in various settings. Functional and non-functional tests can provide valuable information for identifying setup and task-related VCs. For instance, tests can involve varying the input files, their sizes, formats, and exploring different parameter configurations. These tests can help identify resource availability requirements (e.g., memory or number of cores required by the nodes or cluster) and configuration parameter constraints (e.g., valid range or format). It is important to note that this assumes the user has access to nodes with large capacity to identify task memory consumption for large input files or tasks with greedy processing.

Workflow execution traces. If the workflow system has the capability to capture execution traces, also known as workflow provenance, then these traces can be collected during the debugging and testing phases of the workflow. These traces can serve as valuable resources for refining existing constraints, such as resource availability and configuration parameters, and even for identifying new constraints. Specifically, by analyzing workflow traces from multiple executions, it is possible to extract information such as the minimum memory requirement for a particular task. Moreover, execution traces can be leveraged to learn correlations or functions, which can be used, for example, to predict the memory needed by a node to perform a task correctly based on the size of the input files. This will enable the specification of dynamic VCs that are associated with the workflow task. Execution traces can also be utilized in specifying metamorphic relations. By mining the traces, it may become possible to learn functions that describe the relationships between input and output files for a given task. Furthermore, execution traces can be employed to set a maximum time limit for task execution.

If a substantial number of execution traces is accessible, then it may be possible to acquire more advanced VCs. For instance, it may become feasible to predict the quality or precision of workflow results based on its input, as exemplified in a material science user story. If users annotate the quality of results obtained during tests and specify the inputs responsible for such quality, correlations can be explicitly drawn or even learned.

VCs reuse across workflows. Developers often do not specify a workflow from scratch but reuse existing workflows as building blocks or modify and repurpose them. They may also use workflow tasks that have already been used in existing workflows. In such cases, some VCs associated with the reused workflow can be used in the context of the newly developed workflow. Task-related VCs, such as the availability of the executable, resource requirements, configuration parameters, license, and task limits, as well as metamorphic relations, can be reused in certain cases. However, it should be noted that the newly developed workflow may exhibit different features or requirements, which may necessitate adjustments to the reused constraints. For example, a task-end limit may be greater or lower depending on the properties of the input files used in the newly developed workflows.

We have discussed, in this section, various sources of information that developers can utilize to draw VCs. From the above discussion, it becomes apparent that VCs' specification should not be left until the end of workflow development, as suggested in the introduction of this section, but should, instead, be an integral part of the workflow development cycle, whereby VCs are specified and refined gradually during the debugging and testing operations. It is also worth noting that certain VCs are somewhat generic and apply to any data analysis workflow, making their specification straightforward. This is the case, for example, for constraints ensuring infrastructure health and the validity of task licenses.

Workflow repair. The evaluation of our VCs can lead to three different actions: None, stopping an execution with a defined error message, or only issuing a warning into the workflow logs. However, one could also think of other possible reactions to a broken constraint. One intriguing idea is to try to automatically 'repair' the ongoing workflow execution based on the specific constraint [76]. For instance, a VC that fails because the minimal memory requirement of a task is not fulfilled on the foreseen node could also report its requirements to the scheduler to instruct it restarting the task on a more powerful node. A constraint testing the availability of a valid software license could also try to acquire such a license from on online repository; VCs failing due to ill-formatted input files could try reformatting these files. Such repairs probably are not feasible in all cases described in Table 2; for instance, a missing input file points to a problem that would be hart to cure without an in-depth evaluation of the workflow's history. We consider an extension of VCs into rules with a more diverse universe of consequences than just 'true' or 'false' as a particularly interesting route for future work.

9. Conclusions

In this article, we introduced VCs as a means to make implicit assumptions in data analysis workflows explicit, allowing a workflow engine to perform fine-grained status checking and to take proper action if needed. We defined a formal model connecting VCs to the core elements of DAWs, namely tasks for computations, files for data exchanges, and nodes for execution. Based on this formal model, we introduced different types of VCs and classified them according to six dimensions. We extensively discussed related concepts in various fields of research to show that (a) VCs indeed are a vital and ubiquitous concept, but at the same time, (b) a unifying theory was missing, and (c) support for VCs should be considered as partial at best in production or research systems. We hope our work will help to improve this situation by making VCs an integral part of future DAW languages and systems. VCs can support debugging, save energy and time by an early failure of workflow executions, provide traceable warnings or error

messages, and raise confidence in analysis results as they help making DAWs more reliable.

Several extensions to our work are possible. We discussed sharing of VCs and their extension to workflow repair in Section 8. In certain situations, e.g., IoT, DAWs are increasingly often used to analyze data streams, which would pose specific requirements to validity checking and require fundamentally changing their semantics; for instance, the notion of failure would need to be revisited. One could also increase the expressiveness of VCs by allowing constraints that affect groups of tasks (e.g., the total memory of a group of tasks scheduled on a node may not exceed the overall memory of the node) or groups of files (e.g., the files sent to different downstream tasks must be identical). VC checking could directly link to counter actions; for instance, breaking a constraint about necessary memory on a node could result in feedback to the scheduler and trigger a re-scheduling of affected tasks. Another idea worth exploring is an analysis of trace files of (failed) workflow executions to automatically find VCs by studying the common attributes of an execution leading to a certain behavior, similar to provenance patterns [77] or methods for failure predictions [78]. We leave such ideas for future work.

CRediT authorship contribution statement

Florian Schintke: Conceptualization, Project administration, Visualization, Writing – original draft, Writing – review & editing. **Khalid Belhajjame:** Writing – review & editing. **Ninon De Mecquenem:** Writing – original draft. **David Frantz:** Writing – original draft. **Vanessa Emanuela Guarino:** Writing – original draft. **Marcus Hilbrich:** Writing – original draft, Writing – review & editing. **Fabian Lehmann:** Writing – original draft. **Paolo Missier:** Writing – review & editing. **Rebecca Sattler:** Writing – original draft. **Jan Arne Sparka:** Writing – original draft. **Daniel T. Speckhard:** Writing – original draft. **Hermann Stolte:** Writing – original draft. **Anh Duc Vu:** Writing – original draft. **Ulf Leser:** Conceptualization, Funding acquisition, Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Florian Schintke, Ninon De Mecquenem, Marcus Hilbrich, Fabian Lehmann, David Frantz, Vanessa Emanuela Guarino, Rebecca Sattler, Jan Arne Sparka, Daniel T. Speckhard, Hermann Stolte, Anh Duc Vu, Ulf Leser report financial support was provided by German Research Foundation (CRC 1404 'FONDA', project 414984028). Daniel T. Speckhard reports financial support was provided by IMPRS for Elementary Processes in Physical Chemistry. Daniel T. Speckhard reports financial support was provided by Max Planck Graduate Center for Quantum Materials.

Data availability

No data was used for the research described in the article.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as CRC 1404, project 414984028. D.S. acknowledges support by the IMPRS for Elementary Processes in Physical Chemistry and support from the Max Planck Graduate Center for Quantum Materials.

References

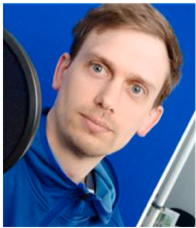
[1] R.F. da Silva, et al., A characterization of workflow management systems for extreme-scale applications, Future Gener. Comput. Syst. 75 (2017) 228–238.
[2] C.S. Liew, et al., Scientific workflows: Moving across paradigms, ACM Comput. Surv. 49 (4) (2017) 66:1–66:39.
[3] S.C. Boulakia, et al., Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities, Future Gener. Comput. Syst. 75 (2017) 284–298.
[4] J. Janssen, et al., Pyiron: An integrated development environment for computational materials science, Comput. Mater. Sci. 163 (2019) 24–36.
[5] C. Witt, J. van Santen, U. Leser, Learning low-wastage memory allocations for scientific workflows at IceCube, in: Int. Conf. on High Performance Computing & Simulation, HPCS, IEEE, 2019, pp. 233–240.
[6] O.V. Sukhoroslov, Toward efficient execution of data-intensive workflows, J. Supercomput. 77 (8) (2021) 7989–8012.
[7] C. Witt, D. Wagner, U. Leser, Feedback-based resource allocation for batch scheduling of scientific workflows, in: Int. Conf. on High Performance Computing & Simulation, HPCS, IEEE, 2019, pp. 761–768.
[8] J. Yu, R. Buyya, K. Ramamohanarao, Workflow scheduling algorithms for Grid computing, in: F. Xhafa, A. Abraham (Eds.), Metaheuristics for Scheduling in Distributed Computing Environments, Springer Berlin Heidelberg, 2008, pp. 173–214.
[9] M. Hilbrich, et al., A consolidated view on specification languages for data analysis workflows, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering, ISoLA, in: Lecture Notes in Computer Science, Vol. 13702, Springer, 2022, pp. 201–215.
[10] R.F. da Silva, et al., A community roadmap for scientific workflows research and development, in: Workshop on Workflows in Support of Large-Scale Science, WORKS, IEEE, 2021, pp. 81–90.
[11] U. Radetzki, et al., Adapters, shims, and glue - service interoperability for in silico experiments, Bioinformatics 22 (9) (2006) 1137–1143.
[12] T.M. Oinn, et al., Taverna: lessons in creating a workflow environment for the life sciences, Concurr. Comput. Pract. Exp. 18 (10) (2006) 1067–1100.
[13] S. Kanwal, et al., Investigating reproducibility and tracking provenance - A genomic workflow case study, BMC Bioinform. 18 (1) (2017) 337.
[14] F. Lehmann, et al., FORCE on nextflow: Scalable analysis of earth observation data on commodity clusters, in: G. Cong, M. Ramanath (Eds.), 1st Int. Workshop on Complex Data Challenges in Earth Observation, in: CEUR Workshop Proceedings, vol. 3052, CEUR-WS.org, 2021.
[15] C. Schiefer, et al., Portability of scientific workflows in NGS data analysis: A case study, 2020, CoRR arXiv:2006.03104.
[16] L. Affetti, A. Margara, G. Cugola, FlowDB: Integrating stream processing and consistent state management, in: Int. Conf. on Distributed and Event-Based Systems, DEBS, ACM, 2017, pp. 134–145.
[17] C. Draxl, M. Scheffler, The NOMAD laboratory: from data sharing to artificial intelligence, J. Phys.: Mater. 2 (3) (2019) 036001.
[18] E. Engel, R.M. Dreizler, Density functional theory, Theoret. Math. Phys. (2011) 351–399.
[19] M. Scheffler, et al., FAIR data enabling new horizons for materials research, Nature 604 (7907) (2022) 635–642.
[20] C.W. Andersen, et al., OPTIMADE, an API for exchanging materials data, Sci. Data 8 (1) (2021) 217.
[21] A. Gulans, et al., Exciting: a full-potential all-electron package implementing density-functional theory and many-body perturbation theory, J. Phys.: Condens. Matter 26 (36) (2014) 363202.
[22] V. Blum, et al., The FHI-aims code: All-electron, ab initio materials simulations towards the exascale, 2022, CoRR abs/2208.12335. arXiv:2208.12335.
[23] J. Hafner, Ab-initio simulations of materials using VASP: Density-functional theory and beyond, J. Comput. Chem. 29 (13) (2008) 2044–2078.
[24] T. Vogel, et al., Challenges for verifying and validating scientific software in computational materials science, in: Int. Workshop on Software Engineering for Science (SE4Science), IEEE, 2019, pp. 25–32.
[25] A. Buccheri, et al., Excitingtools: An exciting workflow tool, J. Open Sour. Softw. 8 (85) (2023) 5148.
[26] F. Caruso, D. Novko, C. Draxl, Photoemission signatures of nonequilibrium carrier dynamics from first principles, Phys. Rev. B 101 (3) (2020) 035128.
[27] C. Carbogno, et al., Numerical quality control for DFT-based materials databases, Npj Comput. Mater. 8 (1) (2022) 1–8.
[28] D.T. Speckhard, et al., Extrapolation to complete basis-set limit in density-functional theory by quantile random-forest models, 2023, CoRR arXiv:2303.14760.
[29] D. Frantz, FORCE—Landsat + Sentinel-2 analysis ready data and beyond, Remote Sens. 11 (9) (2019) 1124.
[30] M. Diaz, Petri Nets: Fundamental Models, Verification and Applications, John Wiley & Sons, 2009.

[31] W.M. Johnston, J.R.P. Hanna, R.J. Millar, Advances in dataflow programming languages, ACM Comput. Surv. 36 (1) (2004) 1–34.
[32] J. Sroka, et al., A formal semantics for the Taverna 2 workflow model, J. Comput. System Sci. 76 (6) (2010) 490–508.
[33] D. Zinn, et al., Scientific workflow design with data assembly lines, in: E. Deelman, I.J. Taylor (Eds.), Workshop on Workflows in Support of Large-Scale Science, WORKS, ACM, 2009.
[34] E.A. Lee, A.L. Sangiovanni-Vincentelli, A framework for comparing models of computation, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 17 (12) (1998) 1217–1229.
[35] F. Kastrati, G. Moerkotte, Generating optimal plans for Boolean expressions, in: Int. Conf. on Data Engineering, ICDE, IEEE Computer Society, 2018, pp. 1013–1024.
[36] H. Garcia-Molina, Database Systems: The Complete Book, Pearson Education, 2014.
[37] P.W.P.J. Grefen, P.M.G. Apers, Integrity control in relational database systems - an overview, Data Knowl. Eng. 10 (1993) 187–223.
[38] E.M. Clarke, T.A. Henzinger, H. Veith, Introduction to model checking, in: E.M. Clarke, et al. (Eds.), Handbook of Model Checking, Springer, 2018, pp. 1–26.
[39] D. Beyer, S. Gulwani, D.A. Schmidt, Combining model checking and data-flow analysis, in: E.M. Clarke, et al. (Eds.), Handbook of Model Checking, Springer, 2018, pp. 493–540.
[40] W.M.P. van der Aalst, M.L. Rosa, F.M. Santoro, Business process management - don't forget to improve the process!, Bus. Inf. Syst. Eng. 58 (1) (2016) 1–6.
[41] M. Dumas, et al., Advanced process modeling, in: Fundamentals of Business Process Management, Springer, Berlin, Heidelberg, 2013, pp. 97–153.
[42] B. Meyer, Eiffel: A language and environment for software engineering, J. Syst. Softw. 8 (3) (1988) 199–246.
[43] V.R. Pratt, Semantical considerations on Floyd-Hoare logic, in: 17th Ann. Symp. on Foundations of Computer Science (Sfcs 1976), IEEE, 1976, pp. 109–121.
[44] H. Wasserman, M. Blum, Software reliability via run-time result-checking, J. ACM 44 (6) (1997) 826–849.
[45] S. Klabnik, C. Nichols, The Rust Programming Language (Covers Rust 2018), No Starch Press, 2019.
[46] W. Tan, M. Zhou, Business and Scientific Workflows: A Web Service-Oriented Approach, John Wiley & Sons, Ltd, 2013.
[47] R. Milner, Communicating and Mobile Systems: the Pi-Calculus, Cambridge University Press, 1999.
[48] W.J. Fokkink, Introduction to Process Algebra, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2000.
[49] C. Baier, J. Katoen, Principles of model checking, MIT Press, 2008.
[50] D. Baylor, et al., TFX: A TensorFlow-based production-scale machine learning platform, in: SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, ACM, 2017, pp. 1387–1395.
[51] M. Zaharia, et al., Accelerating the machine learning lifecycle with MLflow, IEEE Data Eng. Bull. 41 (4) (2018) 39–45.
[52] E. Bisong, Kubeflow and Kubeflow pipelines, in: Building Machine Learning and Deep Learning Models on Google Cloud Platform, Apress Berkeley, CA, 2019, pp. 671–685.
[53] P. Amstutz, et al., Common Workflow Language, v1.0, 2016.
[54] M.R. Crusoe, et al., Methods included: Standardizing computational reuse and portability with the common workflow language, Commun. ACM 65 (6) (2022) 54–63.
[55] P. Di Tommaso, et al., Nextflow enables reproducible computational workflows, Nature Biotechnol. 35 (4) (2017) 316–319.
[56] J. Köster, S. Rahmann, Snakemake – a scalable bioinformatics workflow engine, Bioinformatics 34 (20) (2018) 3600.
[57] B.P. Harenslak, J. de Ruiter, Data Pipelines With Apache Airflow, Simon and Schuster, 2021.
[58] M. Zaharia, et al., Apache Spark: a unified engine for big data processing, Commun. ACM 59 (11) (2016) 56–65.
[59] P. Carbone, et al., Apache Flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (4) (2015) 28–38.
[60] T. Ben-Nun, et al., Workflows are the new applications: Challenges in performance, portability, and productivity, in: P3HPC@SC, IEEE, 2020, pp. 57–69.
[61] E. Deelman, et al., Workflows and e-science: An overview of workflow system features and capabilities, Future Gener. Comput. Syst. 25 (5) (2009) 528–540.
[62] M. Rynge, et al., Integrity protection for scientific workflow data: Motivation and initial experiences, in: T.R. Furlani (Ed.), Pract. and Exp. in Advanced Research Computing on Rise of the Machines (Learning), PEARC, ACM, 2019, pp. 17:1–17:8.
[63] Y. Gil, et al., Mind your metadata: Exploiting semantics for configuration, adaptation, and provenance in scientific workflows, in: L. Aroyo, et al. (Eds.), The Semantic Web - ISWC, in: Lecture Notes in Computer Science, Vol. 7032, Springer, 2011, pp. 65–80.
[64] A.L. Lamprecht, et al., Perspectives on automated composition of workflows in the life sciences, F1000Research 10 (897) (2021).
[65] A. Lamprecht, User-Level Workflow Design - A Bioinformatics Perspective, Lecture Notes in Computer Science, vol. 8311, Springer, 2013.
[66] Y. Gil, et al., Wings: Intelligent workflow-based design of computational experiments, IEEE Intell. Syst. 26 (1) (2011) 62–72.
[67] J.C. Ison, et al., EDAM: an ontology of bioinformatics operations, types of data and identifiers, topics and formats, Bioinformatics 29 (10) (2013) 1325–1332.
[68] J. Qin, T. Fahringer, Scientific Workflows: Programming, Optimization, and Synthesis with ASKALON and AWDL, Springer, 2012.
[69] X. Liu, Y. Yang, J. Chen, Temporal QoS Management in Scientific Cloud Workflow Systems, Elsevier, 2012.
[70] E. Deelman, et al., The future of scientific workflows, Int. J. High Perform. Comput. Appl. 32 (1) (2018) 159–175.
[71] L. Moreau, A canonical form for PROV documents and its application to equality, signature, and validation, ACM Trans. Internet Technol. 17 (4) (2017) 35:1–35:21.
[72] A.D. Vu, et al., Contract-driven design of scientific data analysis workflows, in: E-Science 2023, IEEE, 2023, pp. 1–10.
[73] A.D. Vu, et al., Design by contract revisited in the context of scientific data analysis workflows, in: E-Science 2023, IEEE, 2023, pp. 1–2.
[74] M. Fähndrich, M. Barnett, F. Logozzo, Embedded contract languages, in: S.Y. Shin, et al. (Eds.), Symp. on Applied Computing, SAC, ACM, 2010, pp. 2103–2110.
[75] M. Fowler, Domain-Specific Languages, Addison-Wesley Professional, 2010.
[76] C.L. Goues, M. Pradel, A. Roychoudhury, Automated program repair, Commun. ACM 62 (12) (2019) 56–65.
[77] Z. Miao, et al., Going beyond provenance: Explaining query answers with pattern-based counterbalances, in: Proceedings of the 2019 International Conference on Management of Data, SIGMOD '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 485–502.
[78] F. Salfner, M. Lenk, M. Malek, A survey of online failure prediction methods, ACM Comput. Surv. 42 (3) (2010) 10:1–10:42.

Florian Schintke heads the 'Distributed Algorithms' research department at Zuse Institute Berlin (ZIB). He received his Diploma in computer science from the Technical University Berlin with distinction in 2000 and then joined ZIB's computer science group headed by Prof. Alexander Reinefeld as a research associate. He received his doctoral degree from Humboldt University Berlin in 2010. His current research interests include distributed algorithms and distributed data management in particular fault-tolerance, scalability, big data, resource management, modern networks, and scientific workflows.

Khalid Belhajjame is an associate professor with HDR (habilitation to direct research) at the Parie Dauphine University, where he is a member of the LAMSADE research laboratory. His research focuses on information and knowledge management. In particular, he has made significant contributions in the areas of data preparation, data privacy and protection, eScience, scientific workflow management, provenance tracking and exploitation, and knowledge graphs. He is currently a member of the steering committee of the GDR MaDICS, a national research network for interdisciplinary data science research, co-leader of the Database Working Group for Remote Sensing Data in the Earth Science Informatics (IEEE GRSS), and section editor of the Elsevier MethodX journal.

Ninon De Mecquenem is a Ph.D. student in Computer Science in Ulf Leser's group at the Knowledge Management in Bioinformatics Lab at the Humboldt University in Berlin. She holds a bachelor's degree in biology from the University of Perpignan and a master's degree in bioinformatics from the University of Bordeaux. She worked as a data scientist in the private sector for two years before starting her PhD. She is now working on rewriting bioinformatics workflows for FONDA's A2 group.



David Frantz is an Assistant Professor at Trier University, where he leads the 'Geoinformatics - Spatial Data Science' department. He earned his Diploma in Applied Environmental Sciences and obtained his Ph.D. in Remote Sensing from Trier University. From 2017 to 2021, he was a Postdoctoral Researcher at Patrick Hostert's 'Earth Observation Lab' at Humboldt-Universität zu Berlin, collaborating on interdisciplinary projects with social ecologists and computer scientists. His research focuses on translating raw Earth observation data into customized information to address a diverse array of environmental research and monitoring requirements.

Vanessa Emanuela Guarino is a Ph.D. candidate affiliated with the Kainmüller Lab (Laboratory of Biomedical Image Analysis) at the Max-Delbrück-Center for Molecular Medicine Berlin and is currently funded by FONDA, a collaborative research center of the German Research Foundation (DFG). Her background ranges between financial engineering and statistics and her work focuses on principle ways to construct neural network architectures. In particular, her projects aim at analyzing and defining the mathematical foundations of deep learning, in order to render models invariant to small perturbations and aware of their degree of uncertainty.

Marcus Hilbrich is the Scientific Coordinator of FONDA. FONDA aims to improve software usage, creation, and reuse by improving workflows. Doctorate engineer Marcus Hilbrich supports FONDA as a software engineer based on research on the management of software artifacts, e.g., based on ßMACH, a method to guide software artifact management. In addition, he provides knowledge of different computer science disciplines, like HPC, clouds, grids, security, performance modeling, management, and teaching. He supported a diverse set of projects in academics and industry at the Technische Universität Dresden, Universität Paderborn, Technische Universität Chemnitz, and currently Humboldt-Universität zu Berlin.

Fabian Lehmann is a Ph.D. student in computer science and has been a member of Ulf Leser's group at the Knowledge Management in Bioinformatics Lab at the Humboldt-Universität zu Berlin since 2020. He received his Bachelor's and Master's degrees from TU Berlin between 2015 and 2020 and is currently funded through FONDA, a Collaborative Research Center of the German Research Foundation (DFG). His research focuses on improving the execution of distributed workflows, with a particular emphasis on scheduling and data management.

Paolo Missier is a Professor of Scalable Data Analytics with the School of Computing at Newcastle University, where he leads the Scalable Computing group, and a Fellow (2018–2023) of the Alan Turing Institute, UK's National Institute for Data Science and Artificial Intelligence. His background is in data management and "e-science", i.e., scalable workflow management in support of science. His notable contributions are in the area of data provenance, including the W3C PROV provenance model and provenance for data science. He has recently been focusing on methods for streamlining data engineering for Data Science and AI for Healthcare. He has been (2016–2023) Sr. Associate Editor for the ACM Journal on Data and Information Quality (JDIQ).

Rebecca Sattler is currently a Ph.D. student affiliated with the DBIS (Database Systems and Information Management) group within the Department of Computer Science at Humboldt University of Berlin (HUB) and is currently funded through FONDA, a Collaborative Research Center of the German Research Foundation (DFG). In terms of her research focus, Rebecca's interests encompass several areas within the field of computer science. Specifically, she is dedicated to the exploration of event query languages and the development of algorithms for event query discovery. Additionally, her work delves into the domain of event-based validation of workflow execution, showcasing her commitment to advancing knowledge and innovation in these critical areas of study.

Jan Arne Sparka is a Ph.D. student in the department of Computer Science at Humboldt University of Berlin. Since 2020 he is part of the collaborative research center FONDA where his research focuses on improving the reliability of workflow systems using control operations.

Daniel T. Speckhard is a Ph.D. student currently affiliated with the Max Planck Institute for Solid State Physics, Humboldt-Universität zu Berlin, and the NOMAD Laboratory at the Fritz Haber Institute of Max Planck Society and IRIS Adlershof. Their research focuses on developing statistical learning methods to predict solid material properties that bypass expensive density functional theory computations.

Hermann Stolte is a Ph.D. student at the department of Computer Science at Humboldt University of Berlin and member of the Helmholtz Einstein international Berlin research school in data science (HEIBRiDS). His research is centered around uncertainty management in exploratory data analysis, with a particular focus on data-driven anomaly detection techniques in multi-wavelength astronomy using machine learning.

Anh Duc Vu is a Ph.D. student in the department of Computer Science at Humboldt University of Berlin. Since 2020 he is part of the collaborative research center FONDA where his research focuses on the validity and trustworthiness of data centric systems using techniques from software testing and debugging.

Ulf Leser is the chair of Knowledge Management in Bioinformatics at Humboldt-Universität zu Berlin. He is a computer scientist with long-standing experience in interdisciplinary research regarding managing and analyzing scientific data, especially in Biomedicine. His research focuses on data integration, infrastructures for large-scale scientific data analysis, biomedical text mining, and statistical Bioinformatics. His group develops novel algorithms in these fields and applies them in interdisciplinary research projects, especially in cancer research.