HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR INFORMATIK

# Analyzing the Language of Data Analysis Workflows

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

| | |
|---|---|
| eingereicht von: | Thomas Gasteiger |
| geboren am: | 22.08.1988 |
| geboren in: | Ebersberg |
| Gutachter/innen: | Prof. Dr. Lars Grunske |
| | Dr. Marcus Hilbrich |

eingereicht am: .................................... verteidigt am: ....................................

## Abstract

In data-driven research disciplines, data analysis workflows (DAWs) are widely used. By managing and organizing data-driven analysis, they enable reuse, reproducibility, and traceability of analysis results and are therefore important for big data analysis. The variety of DAW specification languages and their integration into complex and heterogeneous computing environments result in limited portability of DAWs. In this thesis, certain DAW-expressing domain-specific languages (DSLs) are analyzed. Models of these languages are mapped to an established metamodel of DAW languages. All mappings of the selected DSLs are feasible and thus form the basis for future comparability across the languages studied. The textual description of each DAW specification DSL is complemented by mappings of the designed language models to the given metamodel in the form of UML 2 class diagrams.

# Contents

# 1 Introduction

According to Atkinson et al. [1] and Liu et al. [2], computational science is confronted with ever increasing amounts of data. Both Atkinson et al. [1] and Liew et al. [2] point out that computer software helps to harness, integrate, and analyze these data sets. In the field of data-driven research disciplines, Liu et al. [3] state that scientific workflows, also referred to as data analysis workflows (DAWs), as explained by Elfaramawy [4], are used to model data operations. Barker and van Hemert [5] explain that DAWs define a set of analytical activities. It is described by Elfaramawy [4] and Miura and Sladoje [6] that DAWs organize and manage data-driven analysis. DAWs enable reuse, reproducibility, and traceability of analysis results, as Leser et al. [7] explain. Dransch et al. [8] claim that DAWs are essential for big data analysis. Stoudt et al. [9] assert that systematic and traceable DAWs should be fundamental in any academic discipline that engages in data-intensive research.
Based on the aforementioned capabilities of DAWs, Atkinson et al. [1] and Liew et al. [2] describe the prevalence of DAWs in virtually all scientific fields. According to Liew et al. [2], they enable the automation and formalization of scientific methods, resulting in increased productivity and improved scientific methods. Liew et al. [2] also explain that the ubiquity of DAWs comes with the complexity and heterogeneity of the computing infrastructures that process such DAWs. In the absence of standardization, Schiefer et al. [10] note that DAWs often do not run in more than one specific environment.
Fowler [11] and Gronback [12] mention that domain-specific languages (DSLs) cover a wide range of applications. Within this domain, there is a subset of scientific and business-related DSLs with the ability to specify DAWs, as Hilbrich et al. explain [13]. According to Crusoe et al. [14], the large number of these DAW specification DSLs limits the portability of workflows between different systems.
To overcome the hurdle of limited portability and translate from one language to another, a generalized model of these DSLs, a metamodel, is needed, as Hilbrich et al. proclaim [13]. Hilbrich et al. [13] and Liew et al. [15] also state that the abstract structure of a metamodel ensures independence from any system running a DAW specification DSL.
In [13], Hilbrich et al. created such a metamodel for DAW specification DSLs by filtering out similarities and differences between multiple DSLs. The motivation for the work in [13] can be traced to the problem of applying software engineering practices across DSLs. Currently, the metamodel is assumed to provide a common foundation for several DAW specification languages.
The purpose of this paper is to investigate whether a number of scientific and business DSLs can be mapped to the existing metamodel in [13]. To achieve this, the given metamodel and its generalized elements are analyzed. Subsequently, selected DSLs are analyzed with respect to inherent elements. With the knowledge of the existing elements in the metamodel and the individual DSLs, it is possible to determine a potential mapping between specialized elements of the given DAW specification DSLs and generalized elements of the metamodel in [13]. The goals of the work relate to a number of DAW specification languages, from which a selection is made.

These languages are listed below.

- *Common Workflow Language* [16]
- *Airflow* [17]
- *Nextflow* [18]
- *StackStorm* [19]
- *Argo* [20]

With the aim mentioned above, the following research questions are formulated:

**RQ 1:** Do DAW specification languages correspond to the metamodel in [13]?

**RQ 1.1:** Which generalized elements are modeled in the metamodel?

**RQ 1.2:** Which language elements are contained in the individual DSLs?

**RQ 1.3:** Is it possible to map all language elements of each DSL to the generalized elements of the metamodel?

A content analysis will be conducted to answer these questions. Specific definitions of basic terms will be noted. In addition, a detailed analysis and mapping of the above DAW specification DSLs will be conducted to answer the research questions. In doing so, the listed languages can be considered as potential instances of the established metamodel of Hilbrich et al. in [13]. The thesis provides a detailed analysis, description, and mapping of DAW languages.It can be considered as a basis for the comparability of elements in different languages.
Chapter 2 establishes the basic concepts for the work. The approach of the analysis and mapping method from DAW specification languages to the metamodel is explained in Chapter 3. This is followed by a detailed analysis and mapping of each of the listed languages in Chapter 4. The lessons learned are recorded in Chapter 5. A discussion of the validity of the results is provided in Chapter 6. Finally, Chapter 7 addresses possible future work and concludes all previous steps.

# 2 Foundations

The following chapters introduce basic concepts related to DAWs, their design, mapping, and implementation. Four concepts are described in more detail.

## 2.1 Domain-Specific Languages

DSLs can be described as computer programming languages that focus on specific domains [11]. A domain can be defined as a particular class of problems, and a DSL represents an optimized language for that class [21]. In other words, a DSL is a "very specific tool for very specific conditions" [21].

There are few sharp boundaries for defining DSLs [11]. The boundaries are not clearly defined, not black and white, but incremental [21]. Nevertheless, there are a number of key elements and classifications that provide an approximation to a definition.

A DSL is a computer programming language that can be simultaneously understood by humans and executed by computers [11]. The meaning of the notation is comprehensible to both [22]. As mentioned earlier, focusing on a particular domain is another feature of DSLs [23]. Thus, they target specific classes of problems [23]. Moreover, the sense of fluency is elementary [11]. It is not just the individual expression, but the composition of expressions that makes these languages [11]. Limited expressiveness, i.e., a minimum set of language features required to support a particular domain, is also a central aspect of DSLs and characterises the difference between DSLs and general-purpose languages (GPLs) [11]. This limitation reduces flexibility [21], but prevents the possibility of making mistakes [11].

DSLs can be classified according to various criteria. A subset of these languages is called internal DSLs. These languages are embedded in GPLs [21] and therefore use only a subset of the features available in GPLs [11]. In contrast, there is a group of external DSLs [11]. These are stand-alone languages that are separate from the respective application language [11].

Since DSLs are specialized languages, a wide range of languages exists for a variety of domains [11, 12].

## 2.2 Data Analysis Workflows

DAWs, also known as scientific workflows [4], can be defined as a set of data processing tasks with the intent to manage and organize data-driven analysis [10, 24]. They provide a systematic method for specifying data analysis [1, 8]. Workflows consist of three basic components - tasks, dependencies, and data resources [2]. Tasks are fundamental to DAWs [24] and are connected by input-output dependencies [10]. They consume input data and produce output data in a prescribed execution order. [4, 24].

Like all other workflows, DAWs can be defined through the use of a workflow definition language [25], in this case a DAW specificaton DSL.

Workflows are often described as graphs, with vertices depicting both tasks and data, and edges depicting control and data flow [2, 24, 26]. DAWs represented as directed

cyclic graphs (DCGs), unlike DAWs represented as directed acyclic graphs (DAGs), have the ability to represent iterations [2, 26].
Scientific workflows can be assigned to different levels of abstraction [27, 26]. At the conceptual level, the workflows are formulated in the language of scientists [26]. It is the most notional level, followed by the so-called abstract level [26]. At this level are workflow graphs called abstract workflows that specify tasks, dependencies, data flow, and control flow [2, 24]. At the concrete or physical level, the actual execution takes place [24]. Tasks from the abstract level are mapped to executable software and produce a concrete workflow [2, 24, 26].
The sharp increase in the volume of data and the increased need for data analysis in science has driven the importance and interest in DAWs [1, 28]. Workflow specification offers a number of benefits [4]. Reuse, reproducibility, and traceability of analytical results can be established through defined workflows [4]. DAWs are essential for big data analysis [8]. Overall, DAWs have a significant impact on scientific analysis processes and represent an accelerator for scientific progress [27].

## 2.3 Metamodel

In essence, a metamodel is a model of a model [29]. In other words, it outlines the basis for the construction of another model [12]. Metamodels consist of statements about models [30] and give details of the specification to which models must conform [15]. Thus, a model is an instance of a metamodel and is expressed by the metamodel [12]. By generalizing a metamodel, a metametamodel can be created.
Furthermore, a metamodel can be considered as a description of the model syntax [15] or directly equated with the term abstract syntax. In the field of modeling languages, a metamodel can be viewed as a model of the abstract syntax of a language [12].
Instances of languages require a metamodel description in order to be transformed or extended [15, 30]. The metamodel provides support for finding differences and similarities between different models [13, 30].

## 2.4 Metamodel of Data Analysis Workflow Languages

This chapter examines the metamodel of data analysis workflow languages established by Hilbrich et al. in [13]. Its elements and the dependencies between these elements are described. The goal of the analysis is to create an understanding of the given metamodel. Figure 1 illustrates the structure of the metamodel with all its elements. In chapters 4.1 to 4.5, specific instances of language models are mapped to the metamodel. Therefore, a solid understanding of the metamodel forms the basis for understanding the mappings performed from each of the selected DAW specification DSLs to the metamodel by Hilbrich et al.

The metamodel represents the abstracted model of the DAW specification languages and is notated as a UML 2 class diagram [13, ch. 3]. A DAW is modeled as a package of generalized and specialized classes.There are three abstract classes that contain several subtypes:

- *Task*
- *Interconnection*
- *Storage*

At an abstract level, a DAW can be viewed as a composite of the three elements above. Briefly, data from the *Storage* element is manipulated by one or more *Tasks*, with *Interconnections* coordinating the data flow and/or control flow [13, ch. 3]. This basic principle is described by Hilbrich et al. from the analysis of various DAW specification languages [13, ch. 3].

The abstract classes *Task*, *Interconnection*, and *Storage* and their subtypes, as well as other metamodel elements, are explained in more detail below. In [13], a Task is defined as "[...] solution or generic solution idea" [13, ch. 3]. The abstract class *Task* has the four subtypes *InlineDescription. Executable*, *Monitor*, and *InterconnectionManipulation.*
*InlineDescription* is defined as a script that is integrated into a *Task* using a programming language to solve parts of the *Task* [13, ch. 3].
*Executable* describes a computer program that can be used to solve a particular problem [13, ch. 3]. Therefore, an *Executable* in the metamodel can be viewed as a self-contained program that is invoked within a DAW. A reference to this external program is required [13, ch. 3].
*Monitor* is a form of *Task* that connects to external services that "[...] usually monitor some kind of data" [13, ch. 3].
*InterconnectionManipulation* is a construct for representing further subtypes of *Task* with the intent of capturing a variety of tasks found in the DAW specification language palette. Again, *InterconnectionManipulation* includes the three subtypes *SkipTask*, *RepeatTask*, and *FullInterconnectionManipulation.* As the two terms indicate, *SkipTask* allows skipping a *Task* within a DAW, and *RepeatTask* allows repeating a *Task* within a DAW.

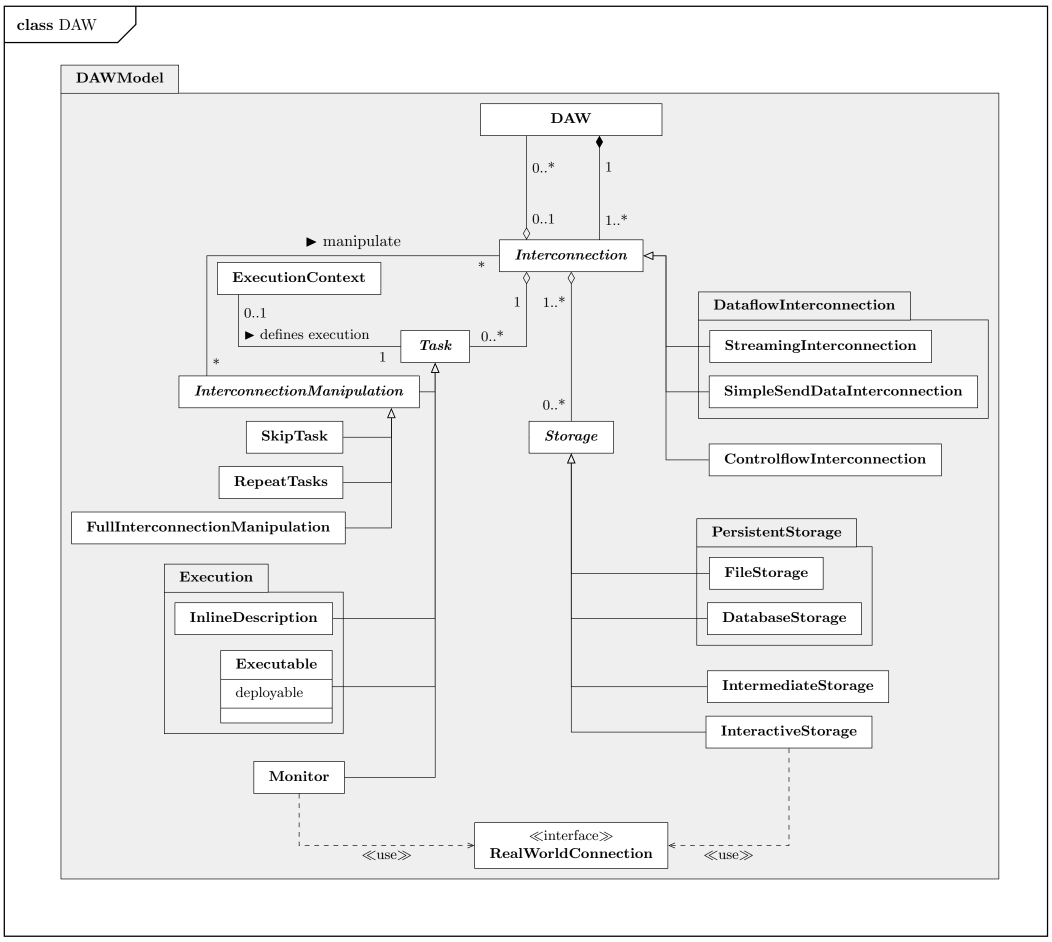

**Fig. 1:** Illustration of the metamodel established by Hilbrich et al. in [13]. The metamodel is expressed as a UML 2 class diagram and contains the three generalized classes *Task*, *Interconnection*, and *Storage*, which can be considered as crucial constructs of any DAW specification language. All UML 2 class diagrams of the modeled languages that appear in the following chapters are mapped to this diagram of the metamodel.

In addition, *FullInterconnectionManipulation* expresses a construct of task processing that is different from any of the other types mentioned. A combination of the above subtypes of *Tasks* is conceivable.

*Interconnections* help define data flow, and/or control flow within a DAW [13, ch. 3]. They are divided into two main categories related to data flow and control flow: *ControlflowInterconnection* and *DataflowInterconnection*.
*ControlflowInterconnection* specifies the order of *Task* executions within a workflow. *DataflowInterconnection* is modeled as a package containing two concrete classes: *StreamingInterconnection* and *SimpleSendDataInterconnection*.
*StreamingInterconnection* describes how to handle a data stream in the course of running *Tasks* [13, ch. 3]. In contrast, *SimpleSendDataInterconnection* contains a data object that is available only after the completion of a *Task* that created this object [13, ch. 3].

The *Storage* element unifies all types of data sources and data sinks under its class [13, ch. 3]. It contains the subclasses of *IntermediateStorage*, *InteractiveStorage*, and *PersistentStorage*
*IntermediateStorage* is the name for data sources and data sinks that can be described as volatile in that they are deleted after a single pass of a DAW.
*InteractiveStorage* is an additional storage concept that deals with arbitrary data from sources and sinks other than *IntermediateStorage* and *PersistentStorage* [13, ch. 3]. *PersistentStorage* is modeled as a package with two subtypes *FileStorage* and *DatabaseStorage*.
*FileStorage* and *DatabaseStorage* cover data storage within a DAW in the form of files and databases stored over the duration of a single DAW run. In a sense, they are the counterpart of *IntermediateStorage*.
The *ExecutionContext* and *RealWorldConnection* classes are elements within the DAW metamodel that are not classified as subtypes of *Task*, *Interconnection*, or *Storage*. *ExecutionContext* describes the possibility to pass data blocks that provide information about concrete computation/execution environments for the execution of specific DAWs [13, ch. 3]. *RealWorldInterface* is the concept that covers "[...] any interaction outside of the DAW system [...]" [13, ch. 3].

Finally, the connections of the classes DAW, *Interconnection*, *Task*, *Storage* and all other classes are briefly discussed. In the package *DAWModel* by Hilbrich et al. the class DAW is directly connected to the class *Interconnection* [13, Figure 3]. From there, the edges lead to the two abstract classes *Task* and *Storage* [13, Figure 3]. Besides that, the class *ExecutionContext* connects directly to the class *Task*, and the class *RealWorldConnection* connects to the *Task's* sub-type *Monitor* as well as to the *Storage's* sub-type *InteractiveStorage* [13, Figure 3]. All subclasses, which can still be found in the metamodel, are connected to their main class via an inheritance relationship.

## 2.5 Glossary

This chapter lists definitions of basic terms that relate to the intended mapping of certain DAW specification DSLs to the metamodel.

**DAW:** Data analysis workflows can be defined as series of data processing tasks with the intention of managing and organizing data-intensive analysis [10, 24].

**DSL:** Domain-specific languages can be described as computer programming languages focused on specific domains while having limited expressiveness [11].

**Generalization:** Generalization is a concept that expresses the hierarchy "[...] between a more general classifier and a more specific classifier" [31] in, inter alia, UML 2 class diagrams. "Thus, the specific classifier inherits the features of the more general classifier" [31].

**Mapping:** Mapping may be defined as the assignment of a defined element to a corresponding distinct element. The term mapping can be found across various fields in computer science [32]. In this work, mapping associates a class integrated into a UML 2 class diagram of a particular DSL with a distinct class in the UML 2 class diagram of a given metamodel.

**Metamodel:** A metamodel is a generalized model of models [29]. In the field of modeling languages, a metamodel can be regarded as a model of the abstract syntax of a language [12].

**Task:** Task is a term to be found in the metamodel of Hilbrich et al. and outlines a solution or solution idea [13]. It conforms to the interchangeable terms CWL *Step* in [16], *Airflow Task* in [17], *Nextflow Process* in [18], *StackStorm Action* in [19], and *Argo Template Definitions* in [20].

**UML 2 class diagram:** An UML 2 class diagram is a concept displaying the structure of a system to be modeled [31]. It describes the existence of classes inhering attributes and operations, as well as association, generalization, and dependency relations across distinct classes [31].

# 3 Methodology

This thesis investigated whether selected DAW specification languages conform to the metamodel of [13]. The steps performed are:

- Analyzing the given metamodel
- Analyzing five selected DAW specification DSLs
- Creating models for the analyzed languages
- Mapping the created models to the metamodel

The term language elements was used to outline components of the languages under study that are part of a DAW. To avoid misunderstandings due to name collisions between languages and the metamodel, all language elements belonging to a particular language have the language name as a prefix in the textual notation. These language elements have been modeled as classes in the generated models.
The metamodel of Hilbrich et al. [13] provided guidance for the description and analysis of each selected DAW specification language. Knowledge of all generalized elements contained in the metamodel points the way to finding conforming elements in the defined languages.
Information about such elements was found in the documentation of the five languages. Documentation available on the websites of each language was consulted almost exclusively. It is suspected that the information obtained from other sources was not detailed and complete enough. Examples of existing workflows were only examined when they appeared in the documentation. Looking at workflow examples alone was avoided because it was likely that they would not capture the entirety of the language elements. It is also believed that deriving overall concepts from specific workflow examples is difficult. Therefore, the content analysis of each of the DAW specification DSLs was based solely on documentation provided by the managing organizations of each language.
The citations in the following chapters indicate the basic reference of the documentation and also name the page, chapter, or path of the noted information within the reference. The documentations were examined with the ulterior motive of filtering out only those elements from a language description that correspond to generalized elements in the metamodel. In order not to lose track, all information about these elements was noted textually. After extracting the elements in question, a UML 2 class diagram [31] was implemented, which linked all recognized elements of a given language.
In this process, the elements were modeled as classes and subclasses. They were related by the UML 2 relation concepts dependency, aggregation, composition, and inheritance [31].
It was obvious to develop all language models in the form of UML 2 class diagrams. Since the metamodel was already modeled as a UML 2 class diagram, it made sense to link the generated models to the metamodel by using the same modeling language. A class diagram of workflow components is shown in [33]. Also, a workflow language metamodel in the form of a UML 2 class diagram is shown in [34]. These examples

underline the suitability of UML 2 class diagrams.
Overall, the instantiated UML 2 class diagrams modeled all subtypes of DAW as packages, as used in the metamodel in [13]. From there, a mapping was performed from a single language diagram to the established metamodel. Since the language diagrams only contain elements that were known to map to elements in the metamodel, this step is a mere formality.
By creating UML 2 class diagrams to model the DAW specification DSLs and mapping them to the metamodel, the correspondence between models and metamodel was established. This approach made it possible to answer the research questions.

# 4 Analysis of DAW Languages

In the upcoming chapters, five selected DAW specifications DSLs are analyzed and mapped to the previously mentioned metamodel of Hilbrich et al. Priorities of the language descriptions are set on concepts and elements that correspond to the metamodel. The following list names the five analyzed DAW specification DSLs:

- *Common Workflow Language*
- *Airflow*
- *Nextflow*
- *StackStorm*
- *Argo*

## 4.1 Common Workflow Language

The *Common Workflow Language* (CWL) is a set of standards for creating and porting workflows [14, p. 2]. Because it is a specification rather than software, it can be ported to any platform that supports CWL [16, ch. 1.3.1]. CWL *Workflows* are modeled as DAGs [35, ch. 3.1], and they are notated in a syntax derived directly from YAML syntax [14, p. 5]. Workflows that conform to the CWL standard must contain objects or arrays of objects with the same syntax [35, ch. 2.2]. In addition, a CWL *Workflow* must contain the elements CWL *Input*, CWL *Step*, and CWL *Output* [16, ch. 2.10]. CWL *Workflow* maps to directly to the DAW element in the metamodel.
A CWL *Workflow*, inherent elements, and a mapping to the metamodel are shown in Figure 2.
In the CWL standard, a CWL *Workflow* is divided into CWL *Steps*, which are connected by input and output dependencies [35, chs. 3.1, 4]. CWL *Steps* refer directly to *Tasks* in the metamodel in [13]. There are three specifications of CWL *Steps* used in CWL [16, ch. 2.10]:

- CWL *Command-Line Tools*
- CWL *Expression Tools*
- CWL *Sub-Workflows*

CWL *Command-Line Tools* are tools that execute commands. They can be executed individually or embedded in a series of CWL *Steps* [16, ch. 2.2]. Since commands can be viewed as language elements, it is possible to map the CWL *Command-Line Tools* to the metamodel's *Executable*. To work, the tools require CWL *Input* and CWL *Output* [16, ch. 2.2].

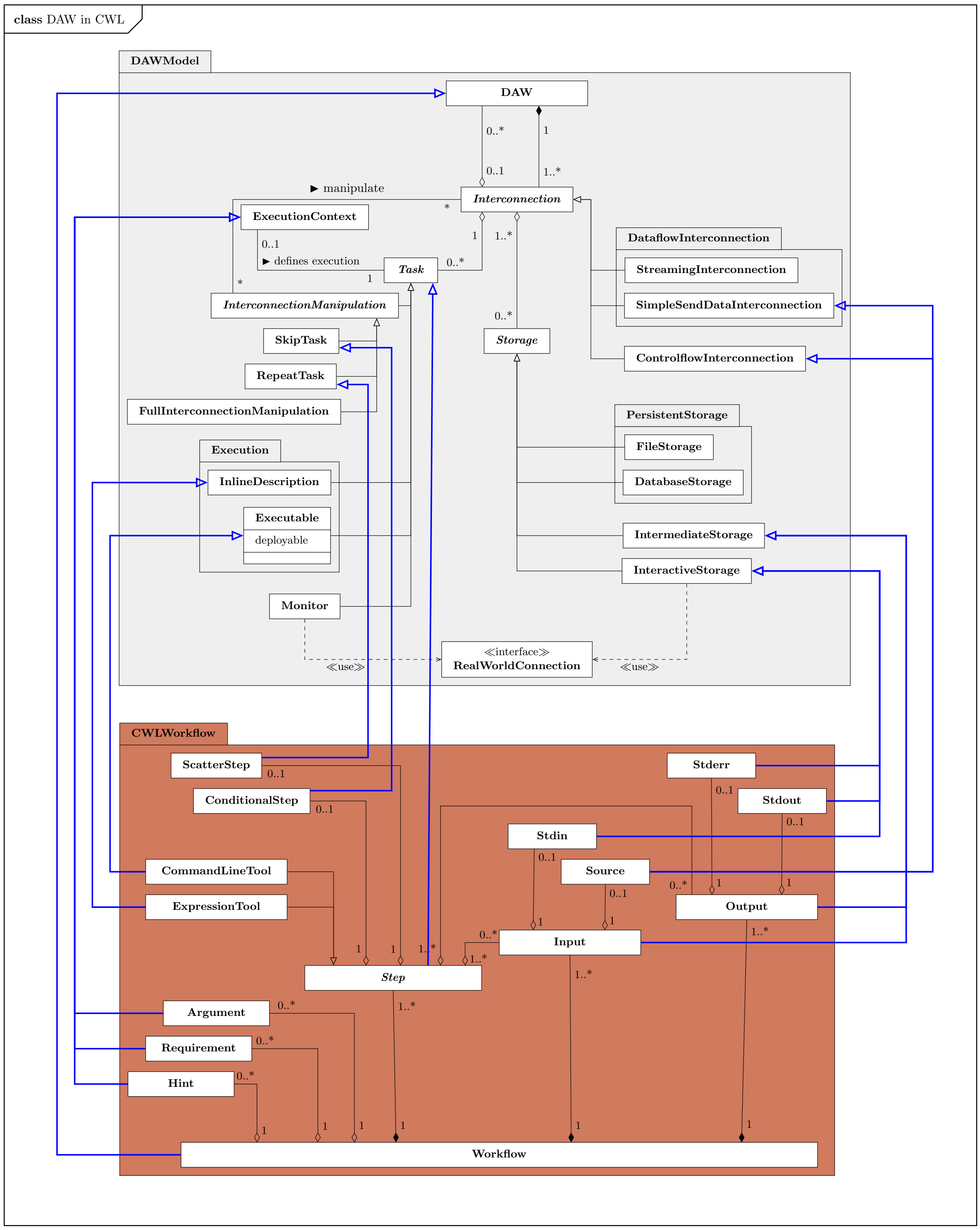

**Fig. 2:** Description of the class diagrams of the metamodel in [13] and a CWL *Workflow* including blue generalization arcs indicating the mapping of CWL *Workflow* elements to specific elements in the metamodel. Some language elements of the CWL *Workflow* are omitted in the diagram because they have no corresponding element in the metamodel.

CWL *Expression Tools* are related to CWL *Command-Line Tools* in terms of the need for CWL *Input* and CWL *Output*. In addition, they can be run alone or as part of a CWL *Workflow* [16, ch. 2.9]. Unlike CWL *Command-Line Tools*, they only execute JavaScript expressions [16, ch. 2.3]. CWL *Expression Tools* map to the *InlineDescription* element in the metamodel due to their use of JavaScript.

In CWL, a CWL *Sub-Workflow*, or nested workflow, is a workflow that is embedded in another CWL *Workflow* [16, ch. 2.10.2]. Therefore, it represents a CWL *Step* to the primary CWL *Workflow* and is similar to the sub-DAWs described in [13].

A CWL *Operation* is a special case of a CWL *Step* and is therefore not listed above. It describes a CWL *Step* that has data links to CWL *Input* and CWL *Output*, but is not specified to be executed [16, ch. 2.11]. Its purpose is to visualize a workflow during development [16, ch. 2.11].

CWL *Inputs* describe the parameters of a CWL *Workflow* or a CWL *Workflow's Step* [16, ch. 2.4.1]. Within the design of CWL, it is possible to implement CWL *Inputs* within CWL *Steps* or at the same level as CWL *Steps* [16, ch. 2.4.1]. They configure the execution of a CWL *Workflow* [16, ch. 2.4.1], and they are compulsory elements in CWL *Workflows* [16, ch. 2.10]. Thereby the input files are read-only [16, ch. 2.4.1]. By calling the optional parameter CWL *Stdin*, CWL *Inputs* can be obtained from the standard input stream [36, ch. 5.1.8].
CWL *Inputs* unify several aspects of the metamodel such as *DataflowInterconnection*, *ControlflowInterconnection* as well as *IntermediateStorage* and *InteractiveStorage*. Several data types are available for CWL *Inputs*. String, int, long, float, double, and null are retrievable primitive types [16, ch. 2.4.1]. In addition, array and record can be listed as complex types. Special types are File, Directory, and Any [16, ch. 2.4.1].

CWL *Outputs* are lists of output parameters that are returned when execution is complete [16, ch. 2.7.1]. The CWL *Output* parameters are either output files or data from the analysis of those output files [16, ch. 2.7.1]. They are compulsory in CWL *Workflows* as CWL *Steps* and CWL *Inputs* [16, ch. 2.10]. It is possible to direct CWL *Output* to standard output or standard error by setting the parameter of CWL *Stdout* or CWL *Stderr* [36, chs. 5.2.1, 5.2.2]. CWL *Outputs* map to *IntermediateStorage* and *InteractiveStorage* in the metamodel.

CWL *Requirements* change the semantics or the runtime environment of a CWL *Workflow* or a CWL *Step* [35, ch. 3.3]. Different CWL *Requirements* can be integrated [35, ch. 3.3]. For example, the CWL *SoftwareRequirement* provides information about the configuration details of the software to be used [36, ch. 5.7]. The CWL *ResourceRequirement* is notable because it specifies hardware resources such as CPU cores, reserved RAM, and reserved filesystem-based memory [36, ch. 5.13]. Several types of CWL *Requirements* are available [35, ch. 3.3] and refer directly to the *ExecutionContext* in the metamodel.

CWL *Hints* are similar to CWL *Requirements*, but have one key difference [35, ch. 3.3]. A CWL *Workflow* can be executed even if the conditions for a CWL *Hint* are not met [35, ch. 3.3].

CWL *Arguments* allow adding more arguments to the CWL *Command Line Tool* [16, ch. 2.5]. This allows environmental information such as hardware and software parameters to be injected into the CWL *Workflow* execution [16, ch. 2.5]. Consequently, the CWL *Arguments* can also be mapped to the *ExecutionContext* element in the metamodel.

Dependencies between *Tasks* in the metamodel, known as *Interconnections*, can be modeled in CWL by using the CWL *Source* parameter [35, ch. 3.1] [16, ch. 2.10.1], which connects CWL *Steps* by cascading a CWL *Output* and a subsequent CWL *Input*. CWL *Steps* in a CWL *Workflow* are not necessarily executed in the order in which they are implemented [16, ch. 2.10.1]. If there are no dependencies between CWL *Steps* and the CWL *Inputs* of each CWL *Step* are prepared [14, p. 7], a parallel run of CWL *Steps* can take place [16, ch. 2.10.1].

It is possible to repeat CWL *Steps* in CWL *Workflows* using CWL *Scattering Steps* [14, p. 7]. Through this feature, a complete CWL *Workflow* or a single CWL *Step* can be executed over a list of CWL *Inputs*, which are understood as arrays [16, ch. 2.10.3]. Here, the number of elements in the arrays determines the number of repetitions [14, p. 8]. It should be noted that the CWL syntax is not appropriate for noting the number of cycles in a CWL *Workflow*, and it is not appropriate to specify conditions to stop a repetition of a cycle [14, p. 8]. The element *RepeatTask*, a sub-subtype of *Tasks* in the metamodel, refers directly to the CWL *ScatterFeatureRequirement*, which is passed as an optional parameter [16, ch. 2.10.3].

*SkipTask*, another feature in the metamodel, corresponds to the element CWL *Conditional Workflow* [16, ch. 2.10.4]. This construct has been available since CWL version 1.2 [14, p. 7]. Using this feature allows a CWL *Step* in a CWL *Workflow* to be skipped by co-signed input parameters [16, ch. 2.10.4]. It is not explicitly documented whether a CWL *Sub-Workflow* can be skipped.

## 4.2 Airflow

*Airflow* is an language, in which *Airflow Workflows* are specified in Python code [17, Overview] and represented as DAGs [17, Core Concepts/Architecture Overview]. Each DAG consists of *Airflow Tasks*, which are basic execution elements of a *Airflow Workflow*, and they are connected by dependencies, that control the execution order of each *Airflow Task* [17, Core Concepts/Architecture Overview]. *Airflow Workflow* corresponds to the element DAW in the metamodel. All this is illustrated in Figure 3, which shows a *Airflow Workflow* and its mapping to the metamodel.
*Airflow Tasks* correspond directly to *Tasks* in the metamodel and can be subdivided into three general types [17, Core Concepts/Architecture Overview]:

- *Airflow Operators*
- *Airflow Sensors*
- *Airflow TaskFlow-Decorated Tasks*

*Airflow Operators* are templates of *Airflow Tasks*, the with the purpose of direct usage [17, Core Concepts/Operators]. They are reusable and require only a few arguments [17, Core Concepts/Operators]. A wide range of these predefined *Airflow Tasks* is provided [17, Core Concepts/Operators]. Examples include a bash command executor, an operator to call arbitrary Python functions, or an operator to send emails [17, Core Concepts/Operators]. Since it is possible to execute Python functions inside *Airflow Operators*, the mapping to *InlineDescription* is given in the metamodel. The mapping from *Airflow Operators* to the *Executable* element in the metamodel results from the option of *Airflow Operators* to execute bash scripts. In addition, various *Airflow Operators* perform the function of *Monitor*, for example, by enabling communication with external services [17, Core Concepts/Operators].

*Airflow Sensors* are subtypes of *Airflow Tasks* that are triggered by events [17, Core Concepts/Sensors]. Thus, *Airflow Sensor* evaluate whether a condition is met and potentialy start interacting [17, Core Concepts/Sensors]. A variety of pre-built *Airflow Sensors* is provided by *Airflow* [17, Core Concepts/Sensors]. Here, the triggers can be based on time, an external event, or file availability [17, Core Concepts/Sensors]. *Airflow Sensors* correspond to *Monitor* and *InlineDescription* in the metamodel.

*Airflow TaskFlow-Decorated Tasks* are versatile *Airflow Tasks* implemented in simple Python code [17, Core Concepts/Architecture Overview]. They can be individual *Airflow Tasks* [17, Core Concepts/TaskFlow]. Likewise, they can be composed of *Airflow Operators* or *Airflow Sensors* [17, Core Concepts/TaskFlow]. For this reason, *Airflow TaskFlow-Decorated Tasks* can execute the *Task* specifications of *InlineDecription*, *Executable*, and *Monitor* in the metamodel.

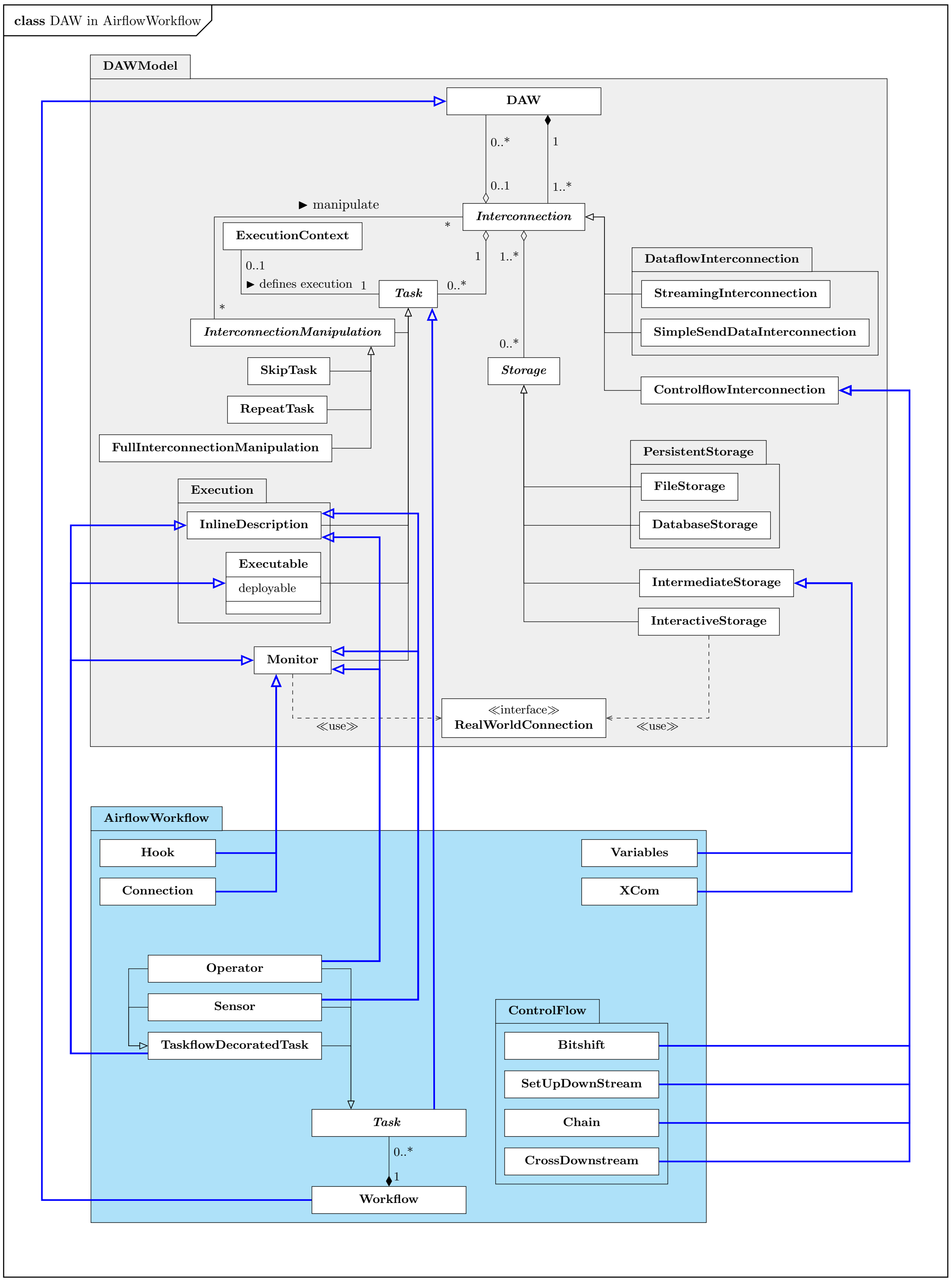

**Fig. 3:** Description of the class diagrams of the metamodel in [13] and a *Airflow Workflow* including blue generalization arcs indicating the mapping of *Airflow Workflow* elements to specific elements in the metamodel. Some language elements of the *Airflow Workflow* are omitted in the diagram because they have no corresponding element in the metamodel.

To create a *Airflow Workflow*, *Airflow Operators*, *Airflow Sensors*, and *Airflow TaskFlow-Decorated Tasks* must be linked together [17, Core Concepts/Architecture Overview]. Therefore, dependencies between each *Airflow Tasks* are required [17, Core Concepts/Architecture Overview]. In a *Airflow DAG*, dependencies can be viewed as edges [17, Controlflow]. The control flow that determines the execution order of each *Airflow Task* is achieved through various methods mentioned below.
First, *Airflow Tasks* can be combined by bit-shift-composition in the form of the operators "<<" and ">>" [17, Core Concepts/Architecture Overview]. The methods known as set_upstream() and set_downstream() also allow *Airflow Tasks* to be combined [17, Core Concepts/Architecture Overview]. Another option is to use the *chain()* method, where tasks can be easily listed in execution order. *Cross_downstream()* describes the method to pair each element of a list with each element of another list [17, Core Concepts/DAGs]. All of the above methods handle control flow in *Airflow* and can be mapped to *ControlflowInterconnection* in the metamodel.

*Airflow XComs* enable communication between *Airflow Tasks* by applying the methods of *xcom_pull()* and *xcom_push()* [17, Core Concepts/XComs]. However, the data transferred by *Airflow XComs* is preserved only for the duration of a single DAG run [17, Core Concepts/XComs]. Therefore, *Airflow XComs* is equivalent to *IntermediateStorage* specified in the metamodel.

*Airflow Variables* form the concept of runtime configuration within *Airflow* [17, Core Concepts/Variables], and it is related to *Airflow XComs* [17, Core Concepts/XComs]. Yet, the stored global key/value pairs of *Airflow Variables* are for an overall configuration [17, Core Concepts/Variables], not for the connection between individual *Airflow Tasks* as in *Airflow XComs* [17, Core Concepts/XComs, Core Concepts/Variables]. It is recommended to use *Airflow Variables* only for values that are runtime dependent [17, Core Concepts/Variables]. Since *Airflow Variables* contains variables that are stored for a single DAW execution, it can be mapped to *IntermediateStorage* of the metamodel.

In *Airflow* there are constructs that make connections to external systems [17, Authoring and Scheduling/Connections & Hooks]. *Airflow Connections* is such a construct and contains a set of parameters [17, Authoring and Scheduling/Connections & Hooks]. These parameters specify, among other things, the type of system that *Airflow* connects to [17, Authoring and Scheduling/Connections & Hooks]. *Airflow Hooks* represents another instance of the above construct. It is a high-level interface for interacting with a system outside of DAW [17, Authoring and Scheduling/Connections & Hooks/Hooks]. Both *Airflow Connections* and *Airflow Hooks* map to *Monitor* of the metamodel.

## 4.3 Nextflow

*Nextflow* is a DAW specification DSL. It uses Linux's built-in command-line and scripting tools [18, Basic concepts]. In *Nextflow*, workflows called *Nextflow Pipeline Scripts* [18, Basic concepts/Processes and channels] are defined as DAGs [18, Tracing & visualisation/DAG visualisation], and they consist of *Nextflow Processes* (nodes) containing *Nextflow Channels* (edges) [18, Basic concepts/Processes and channels]. The concepts of a workflow in *Nextflow* and its mapping to the metamodel are shown in Figure 4. *Nextflow Pipeline Script* map to the element DAW in the metamodel. First, *Nextflow Processes* are defined with *Nextflow Channels* [18, Basic concepts/Processes and channels]. Then, a *Nextflow PipelineScript* is implemented with the composition of the previously noted *Nextflow Processes* [18, Processes].
*Nextflow Processes* map to *Tasks* in the metamodel. Within each *Nextflow Process*, there may be multiple definition blocks [18, Processes]. These blocks are:

- *Nextflow Directives*
- *Nextflow Input*
- *Nextflow Output*
- *Nextflow When Clause*
- *Nextflow Script*

*Nextflow Directives* can optionally configure various settings for the execution of one or more *Nextflow Processes* [18, Processes/Directives]. Among other things, *Nextflow Directives* can configure the requirements of *Nextflow Processes* [18, Processes/Directives]. Consequently, a mapping from *Nextflow Directives* to *ExecutionContext* in the metamodel is possible.

*Nextflow Processes* may contain *Nextflow Channels* that enable communication between *Nextflow Processes* under the data flow programming paradigm [18, Channels]. Due to this fact, *Nextflow Channels* map to *StreamingInterconnection* in the metamodel. *Nextflow Input* and *Nextflow Output* are specifications of *Nextflow Channels* [18, Processes/Inputs, Processes/Outputs]. *Nextflow Stdin* and *Nextflow Stdout* can be used by *Nextflow Input* and *Nextflow Output* [18, Processes/Inputs, Processes/Outputs], resulting in a further mapping to *InteractiveStorage*.

*Nextflow When Clauses* are mechanisms for controlling the execution of a *Nextflow Process* by defining functions that have a boolean return value [18, Processes/When]. They are therefore suitable for skipping the execution of *Nextflow Process*, and consequently correspond to *SkipTask* in the metamodel. However, it must be emphasised that it is better to avoid *Nextflow When Clauses* in *Nextflow Process* blocks and instead use if-statements in the *Nextflow PipelineScript* to increase portability [18, Processes/When].

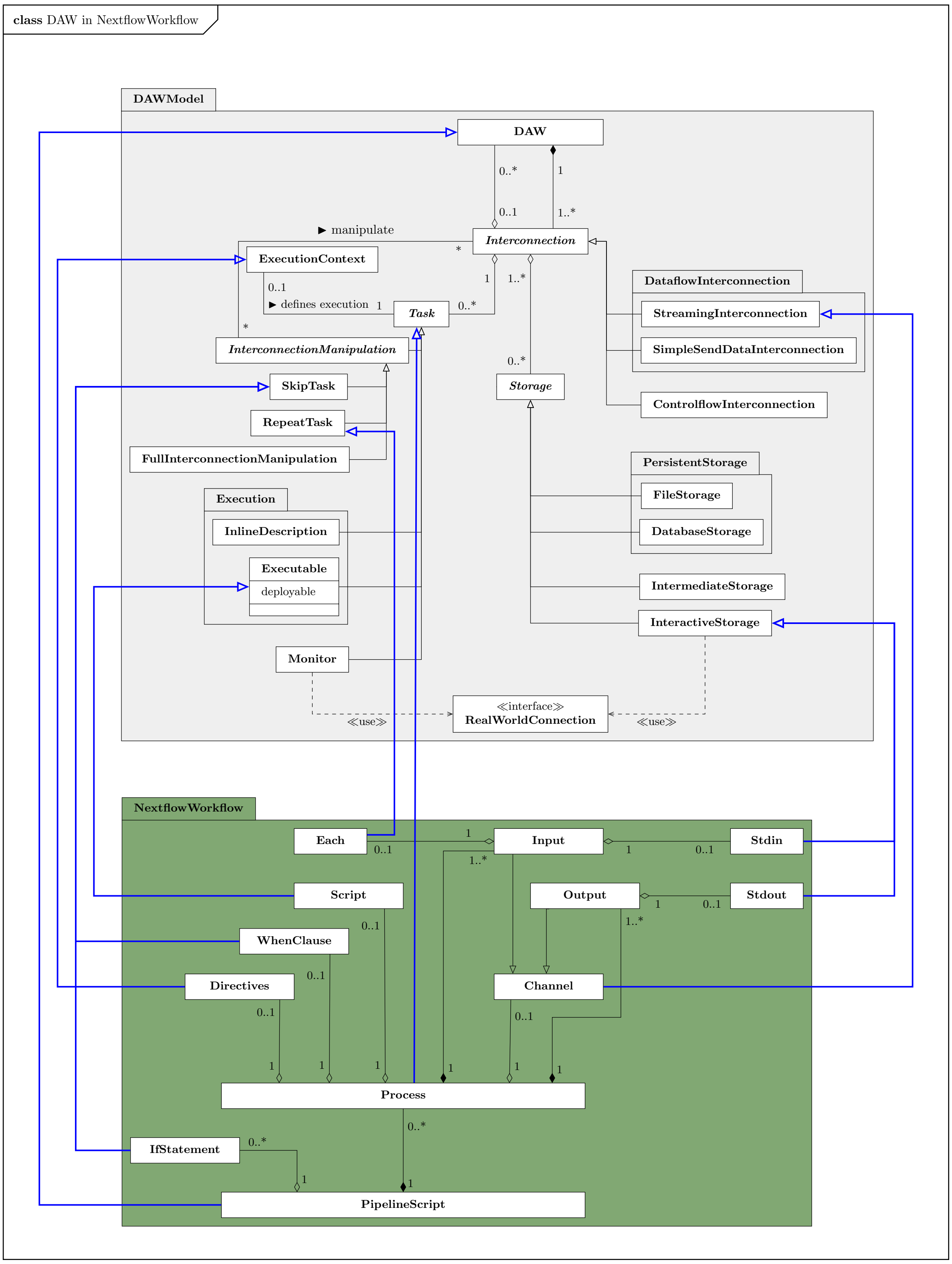

**Fig. 4:** Description of the class diagrams of the metamodel in [13] and a *Nextflow Workflow* including blue generalization arcs indicating the mapping of *Nextflow Workflow* elements to specific elements in the metamodel. Some language elements of the *Nextflow Workflow* are omitted in the diagram because they have no corresponding element in the metamodel.

In *Nextflow*, *Nextflow Processes* can be repeated by using the qualifier *Nextflow Each* in a *Nextflow Input* block [18, Processes/Inputs/Input repeaters]. As long as a new value is received, the *Nextflow Process* including the *Nextflow Input* with the *Nextflow Each* qualifier will be executed repeatedly [18, Processes/Inputs/Input repeaters]. Thus, a *Nextflow Process* containing a *Nextflow Each* qualifier is mapped to the metamodel's *RepeatTask*.

*Nextflow Scripts* determine the blocks in which commands of scripting languages are implemented [18, Basic concepts/Processes and channels]. Within each *Nextflow Process*, any scripting language running on the Linux platform can be executed [18, Basic concepts/Processes and channels]. Scripting languages can also be mixed [18, Processes/Script]. The *Nextflow Script* block in a *Nextflow Process* corresponds to the *Executable* in the metamodel.

## 4.4 StackStorm

*StackStorm* is DAW specification DSL for event-driven workflow automation [19, Getting Started/StackStorm Overview]. It works on the principle of "if an event happens, then respond to that event" [37]. A *StackStorm Workflow* is defined in a YAML file. [19, Automation Basics/Workflows/Orquesta/Orquesta Overview]. In *Orquesta*, the latest and recommended workflow engine available for *StackStorm* [19, Automation Basics/Workflows], *StackStorm Workflows* can be expressed as directed graphs or DCGs and and consists of one or more *StackStorm Tasks* [19, Automation Basics/Workflows/Orquesta/Orquesta Overview]. The element *Orquesta Workflow* corresponds to the element DAW in the metamodel. An illustration of an *Orquesta Workflow*, its elements, and its mapping to the metamodel can be seen in Figure 5. Among others, *StackStorm* describes three basic elements [19, Automation Basics/Actions, Automation Basics/Sensors and Triggers/Sensors, Automation Basics/Rules]:

- *StackStorm Actions*
- *StackStorm Sensors*
- *StackStorm Rules*

*StackStorm Actions* are language elements within *StackStorm* [19, Getting Started/StackStorm Overview] that can perform various automation and remediation tasks [19, Automation Basics/Actions]. Notably, *StackStorm Actions* can be written in any programming language [19, Automation Basics/Actions]. They are mapped in the metamodel to *Tasks*, more specifically, to the *Tasks'* specialization *InlineDescription.*

*StackStorm Sensors* are constructs within *StackStorm* that capture and process events from external systems [37, Automation Basics/Sensors and Triggers/Sensors]. They are written in the Python language [19, Automation Basics/Sensors and Triggers/Sensors]. Collectively, they monitor external systems or respond to incoming events in a defined way and then output *StackStorm Triggers* to *StackStorm* [19, Automation Basics/Sensors and Triggers/Sensors]. In the metamodel, *StackStorm Sensors* correspond to *Monitor* and *InlineDescription.*

*StackStorm Triggers* are compositions of types and optional parameters to identify external events [19, Automation Basics/Sensors and Triggers/Triggers]. *StackStorm Sensors* activate *StackStorm Triggers* that initiate *StackStorm Rules* [19, Automation Basics/Sensors and Triggers/Triggers] to determine *StackStorm Actions* to be invoked [19, Automation Basics/Rules]. Other than that, *StackStorm Timers* are a subset of *StackStorm Triggers* [19, Automation Basics/Rules/Timers]. Their implementation makes it possible to re-execute or skip *StackStorm Actions* based on time intervals or dates and times [19, Automation Basics/Rules/Timers]. Therefore, *StackStorm Timers* correspond to *SkipTask*, *RepeatTask*, and *ExecutionContext* in the metamodel.

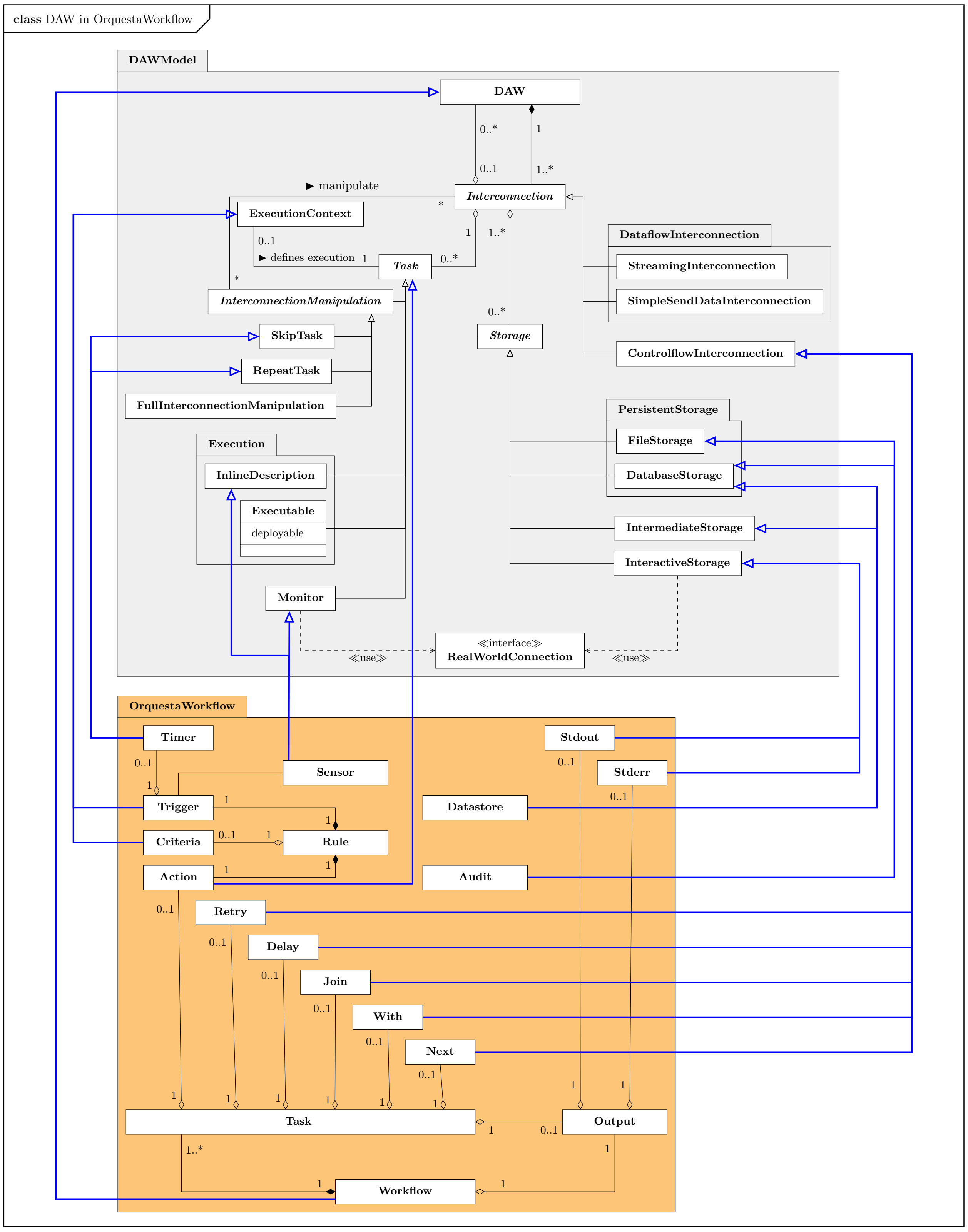

**Fig. 5:** Description of the class diagrams of the metamodel in [13] and a *Orquesta Workflow* including blue generalization arcs indicating the mapping of *Orquesta Workflow* elements to specific elements in the metamodel. Some language elements of the *Orquesta Workflow* are omitted in the diagram because they have no corresponding element in the metamodel.

*StackStorm Rules* are defined in YAML [19, Automations Basics/Rules/Rule Structure] and describe the links between *StackStorm Sensors* and *StackStorm Actions* [37, Automation Basics/Rules]. They map *StackStorm Triggers* injected by *StackStorm Sensors* to *StackStorm Actions* [37, Automation Basics/Rules]. *StackStorm Criteria*, elements injected into *StackStorm Rules* [19, Automation Basics/Rules/Criteria], filter *StackStorm Triggers* [19, Automation Basics/Rules/Trigger], to specify an *StackStorm Action* to be executed [19, Automation Basics/Actions]. *StackStorm Rules* contain *StackStorm Criteria*, *StackStorm Triggers*, and associated *StackStorm Actions* [19, Automation Basics/Rules]. They are assigned to the *ExecutionContext* in the metamodel.

*StackStorm Workflows* map to the DAW element in the metamodel. In *StackStorm*, a distinction is made between two types of *StackStorm Workflows* [19, Automation Basics/Workflows]. An *Orquesta Workflow* can describe complex sequences, while an *ActionChain Workflow* can only notate linear sequences [19, Automation Basics/Workflows]. It is recommended to use the newer and more powerful *Orquesta Workflow* [19, Automation Basics/Workflows]. In the following, all analysis and descriptions refer to *Orquesta Workflows* for the reasons stated above.

*StackStorm Workflows* are constructs of one or more *StackStorm Tasks* chained together [19, Automation Basics/Workflows]. *StackStorm Tasks* are constructs that define which *StackStorm Actions* should be executed with which associated inputs [19, Automation Basics/Workflows]. In addition, they provide instructions about the upcoming sequence after a *StackStorm Task* completes. Consequently, they define the execution sequence in a *StackStorm Workflow* by integrating the attribute *StackStorm Next* [19, Automation Basics/Workflows/Orquesta/Workflow Definition/Task Model].
The *StackStorm Join* attribute can set up a barrier to parallel *StackStorm Task* branches [19, Automation Basics/Workflows/Orquesta/Workflow Definition/Task Model].
Besides, the *StackStorm Delay* attribute allows delaying the execution of a *StackStorm Task.* [19, Automation Basics/Workflows/Orquesta/Workflow Definition/Task Model].
*StackStorm Next* can be extended with the attributes *StackStorm When* and *StackStorm Do* [19, Automation Basics/Workflows/Orquesta/Workflow Definition/Task Model] which attach a condition to the invocation of a task and therefore can be skipped if a condition is not met.
The *StackStorm Retry* attribute allows a *StackStorm Task* to be retried for specific times or a specific *StackStorm Criteria* [19, Automation Basics/Workflows/Orquesta/Workflow Definition/Task Model].
The *StackStorm With* attribute iterates a *StackStorm Action* for each item in a given list [19, Automation Basics/Workflows/Orquesta/Workflow Definition/Task Model].

It must be emphasised that *StackStorm Tasks* do not map uniquely to *Tasks* in the metamodel. Consequently, all of the above attributes map to *ControlflowInterconnection*, not to any form of the metamodel's *Tasks.* For example, mappings to *RepeatTask* or *SkipTask* would be associated with *StackStorm Actions* rather than *StackStorm Tasks.*

In *StackStorm Audit*, information about past *StackStorm Action* executions is stored [37, Advanced Topics/References and Guides/History and Audit]. The context of an event, its *StackStorm Trigger*, the associated *StackStorm Rule* and the resulting *StackStorm Action* are contained [19, Advanced Topics/References and Guides/History and Audit]. *StackStorm Audit* is stored as both a database and an audit log file [19, Advanced Topics/References and Guides/History and Audit]. Therefore, *StackStorm Audit* maps to *FileStorage* and *DatabaseStorage* in the metamodel.

*StackStorm Datastore* stores parameters and associated values in *StackStorm* for reuse by *StackStorm Sensors*, *StackStorm Rules*, and *StackStorm Actions* [19, Automation Basics/Datastore]. By default, all information is stored permanently [19, Automation Basics/Datastore]. However, it is optional to implement a Time to Live (TTL) value to instruct the deletion of items from the *StackStorm Datastore* [19, Automation Basics/Datastore/Setting a Key-Value Pair TTL]. Thus, the *StackStorm Datastore* can be mapped to the *DatabaseStorage* in the metamodel if no TTL is specified. If a TTL is specified, the *StackStorm Datastore* can be mapped to the *IntermediateStorage* in the metamodel, depending on the time frame specified by the TTL value.

## 4.5 Argo Workflows

*Argo Workflows* is an open-source workflow engine [20, Argo Workflows]. Within *Argo Workflows*, it is possible to define each workflow section as a container [20, Argo Workflows]. On the one hand, *Argo Workflow* can be described as a sequence of steps, then again *Argo Workflows* can be modeled as DAGs to illustrate dependencies between steps [20, Argo Workflows]. Figure 6 shows a diagram of a *Argo Workflow* and its mapping to the metamodel.
In general, *Argo Workflows* are specified in the *Argo Workflow.spec* field, which mainly contains lists of *Argo Templates* [20, User Guide/Core Concepts/Workflow Spec] with optional input and output sections [20, Getting Started/Walk Through/The Structure of Workflow Specs]. *Argo Templates* can be informally described as functions that define instructions for execution [20, User Guide/Core Concepts/Workflow Spec]. *Argo Workflow.spec* corresponds to the DAW element in the metamodel.
In contrast, the mapping of *Argo Templates* is more complex, as they are divided into two categories [20, User Guide/Core Concepts/template Types], and they map to a wide variety of elements in the metamodel.

The following describes *Argo Templates* and their mappings. The first category of *Argo Templates*, known as *Argo Template Definitions*, defines the work of each step in an *Argos Workflow* [20, User Guide/Core Concepts/template Types]. Overall, this subset of *Argo Templates* maps to the metamodel's Task.
Next, a selection of specifications of *Argo Template Definitions* that are mapped to the metamodel is described. The selected specifications are:

- *Argo Container*
- *Argo Script*
- *Argo Suspend*
- *Argo Resource*

*Argo Container* templates integrate containers into *Argo Workflows* [20, User Guide/Core Concepts/Container]. Since *Argo Workflows* is designed to integrate containers, *Argo Container* is most commonly used [20, User Guide/Core Concepts/Container]. The purpose of a container suggests that the *Argo Container* template maps to the *Executable* element in the metamodel.

*Argo Script* templates help include scripts by temporarily storing them in files [20, Walk Through/Scripts And Results] and executing them [20, User Guide/Core Concepts/Script]. They allow the use of programming languages and thus comply with the metamodel's *InlineDescription*.

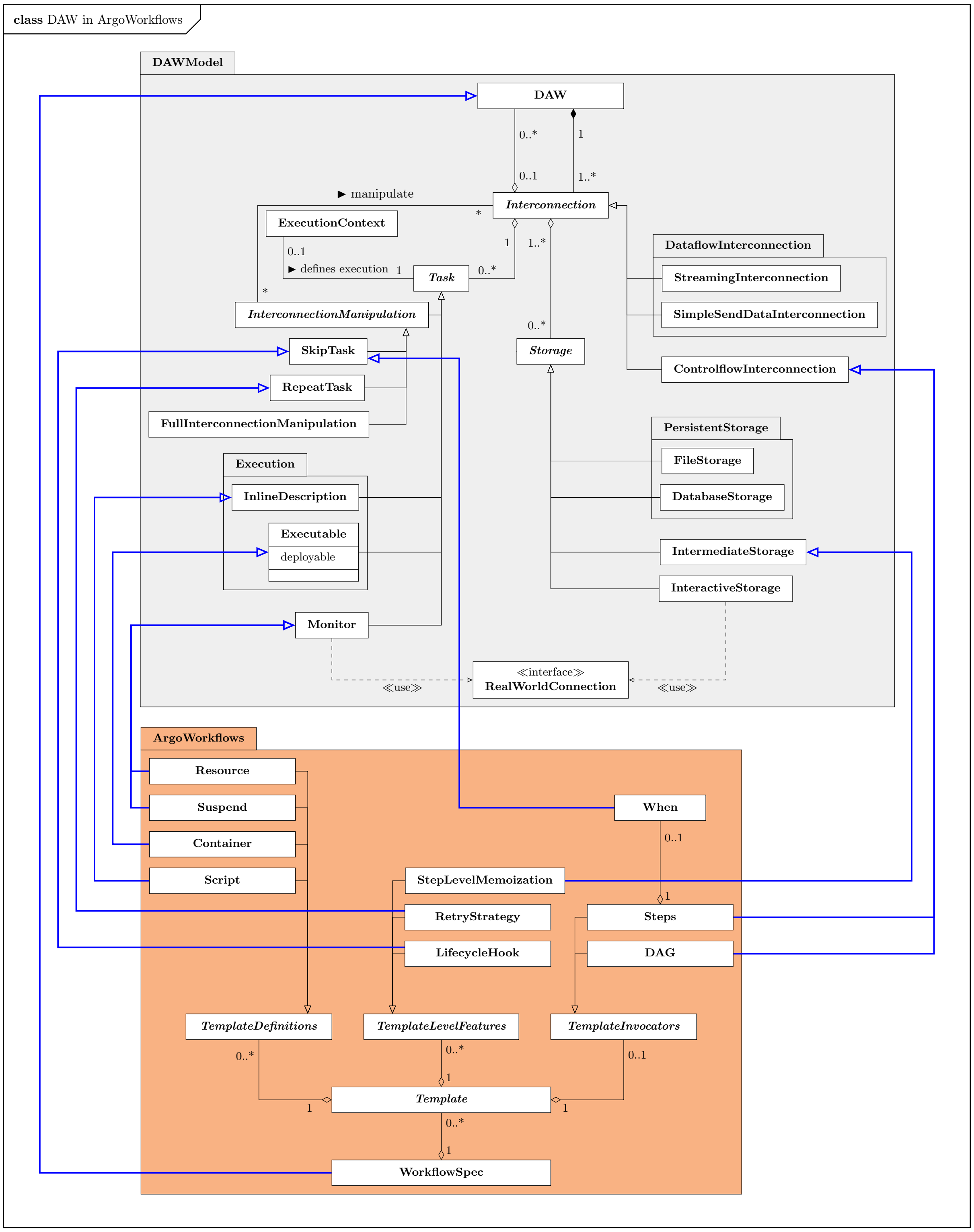

**Fig. 6:** Description of the class diagrams of the metamodel in [13] and a *Argo Workflow* including blue generalization arcs indicating the mapping of *Argo Workflow* elements to specific elements in the metamodel. Some language elements of the *Argo Workflow* are omitted in the diagram because they have no corresponding element in the metamodel.

*Argo Suspend* templates allow a *Argo Workflow* to be suspended [20, User Guide/Core Concepts/Suspend] at a specific node [20, User Guide/UI Features/Intermediate Parameters]. In this way, intermediate parameters can be passed during the run of a *Argo Workflow* via human interaction [20, User Guide/UI Features/Intermediate Parameters]. Consequently, this construct can be mapped to the *Monitor* element in the metamodel. *Argo Resource* templates create connections to cluster resources [20, User Guide/Core Concepts/Resource]. Since these resources are outside of *Argo Workflows* and *Argo Resource* can connect to this external service, it is mapped to the *Monitor* element of the metamodel.

The second category of *Argo Templates* is known as *Argo Template Invocators* [20, User Guide/Core Concepts/Template Invocators]. They facilitate the invocation of other *Argo Templates* and therefore control the execution order [20, User Guide/Core Concepts/Template Invocators]. The two specifications of *Argo Template Invocators* are:

- *Argo Steps*
- *Argo DAG*

The *Argo Steps* template and the *Argo DAG* template, explained in more detail below, both map to the metamodel's *ControlflowInterconnection.*

The *Argo Steps* template is a list of lists and defines the invocation of individual steps containing *Argo Template* in a sequence [20, User Guide/Core Concepts/Steps] to create multi-step *Argo Workflows* [20, User Guide/Core Concepts/Template Invocators, Getting Started/Steps]. The steps can be sequential or parallel [20, User Guide/Core Concepts/Steps]. All individual steps of a *Argo Workflow* are listed in the order they are called in a sequential run. In a parallel run they are noted side by side in an inner list [20, User Guide/Core Concepts/Steps].

As an additional feature, the template *Argo Step* may contain an optional attribute known as *Argo When* that conditionally executes an *Argo Step* [20, User Guide/Core Concepts/Steps]. Derived from this, the *Argo When* attribute maps to *SkipTask* in the metamodel. In addition to the *Argo When* attribute, it is specified that "[...] a wide array of options to control execution [...]" is accessible, next to the *Argo When* attribute. [20, User Guide/Core Concepts/Steps]. However, further documentation of these options is not apparent in the *Argo* documentation.

*Argo DAG* templates use the concept of DAGs to declare dependencies between different tasks, which in turn contain *Argo Templates* [20, User Guide/Core Concepts/DAG]. This can be achieved by including a list of tasks that must be completed before starting the current task [20, User Guide/Core Concepts/DAG]. If there is no dependency, a specific task will be executed immediately [20, User Guide/Core Concepts/DAG]. The term step in the *Argo Step* templates is similar to the term task in the *Argo DAG* template. However, both terms are not further specified in [20] and can only be understood implicitly and refer roughly to the element *Task* in the metamodel.

Another subset of the language elements of *Argo* is called *Argo Features* [20, User Guide/Features], which is explained below. The elements of *Argo Features* that can be mapped to the metamodel are:

- *Argo Template-level Lifecycle Hooks*
- *Argo Retry Strategy*
- *Argo Step Level Memoization*

*Argo Template-level Lifecycle Hooks* initiate actions within *Argo Workflows* when a defined expression is met [20, User Guide/Features/Lifecycle-Hook]. Therefore, a *Argo Template* will not be invoked if an expression is not satisfied. Consequently, the *Argo Template-level Lifecycle Hooks* can be mapped to the *SkipTask* element in the metamodel.

*Argo Retry Strategy* is a construct for retrying failed tasks [20, User Guide/Features/Retries]. Therefore, a mapping to the *RepeatTask* element in the metamodel is feasible. However, an objection can be raised since it is not possible to repeat the entire set of tasks, but only the subset of failed tasks.

*Argo Step Level Memoization* stores output from previous designated *Argo Templates* in a special cache [20, User Guide/Features/Step Level Memoization]. With this strategy, *Argo Workflows* can be optimized by avoiding repetitions of *Argo Templates* [20, User Guide/Features/Step Level Memoization]. The caching concept of the *Argo Step Level Memoization* allows it to be mapped to the *IntermediateStorage* of the metamodel.

## 4.6 Evaluation

In the previous chapters, CWL, *Airflow*, *Nextflow*, *StackStorm*, and *Argo* were analyzed. Mappings to the designated metamodel of Hilbrich et al. were modeled. The different approaches of these DAW specification DSLs and the knowledge of the existence of another variety of such languages [14, 38] illustrate the effort of Hilbrich et al. to establish an abstract, higher-level model in [13].
During the analysis, there was a confrontation with sometimes inconsistent, incomplete, and poorly designed documentation that made it difficult to extract information about the languages and build the resulting models. This problem seems to be more common in the analysis of DAW expressing languages. It has been noted before: "The confusion between workflow languages and enactors is such that the language is not always well defined nor documented" [39].
Already included in the five languages listed above is a wide range of concepts. This spectrum covers everything from data storage to data flow, control flow, and tasks. On this basis, a diverse expression of each model representing a particular DAW specification language is evident. In the sequel, all individual mappings to the metamodel appear in various forms [fig. 2, fig. 3, fig. 4, fig. 5,fig. 6].
Despite all the diversity, it should be noted that the subtypes of the metamodel basic principles *Task*, *Inteconnection*, and *Storage* are found in all five languages. Thus, the language elements can be mapped to generalized elements of the metamodel. In other words, the metamodel does not break, and it is able to handle CWL, *Airflow*, *Nextflow*, *StackStorm*, and *Argo*. The fact that the selected DAW specification DSLs conform to the metamodel is the answer to the research question.

# 5 Lessons Learned

In the previous chapters 4.1 to 4.5, five particular DAW specification DSLs were analyzed. After applying the methodology proposed in Chapter 3, several lessons were learned regarding the chosen approach.
Reviewing the documentation of the selected languages, the same pattern emerged. The most important language elements were found in a simple way. It was possible to note and link them. As soon as more specific language elements were analyzed, the process of classifying, grouping, and noting them slowed down considerably. The volume of language elements presented the challenge of keeping track of the individual DAW specification DSLs.
To solve this problem, a rigorously structured approach to the analysis of language documentation is proposed to be followed at all times. This includes a constant comparison between the generalized elements of the metamodel and the specialized elements of a given language. Precise annotations to the chapters of the documentations are necessary to keep track of the relevant information, because problems with retrieving information occurred.
It can be concluded that it is advisable to complete the analysis of each language in a single coherent process. The concepts and structures of the languages analyzed are very different, and it is difficult to keep track of them simultaneously. In some respects, it is a challenge to follow this procedure, since findings in one language may clarify aspects of another language. Nevertheless, it is beneficial to complete the analysis of each language to the extent possible.
Analyzing the different languages improved the ability to evaluate the documents provided. In the course of time, a firm opinion about the given documentations was formed. It is considered that there is no perfect documentation in the field of the five languages, but that different documentations or different documentation chapters are more understandable and specific than others.
It was concluded that it is complex to write well-written documentation, especially documentation that contains many elements and complex, flexible concepts. Overall, there is the believe that coherent documentation is complete and consistent, and its notation follows a well-thought-out concept. In addition, it contains both textual and graphical representations, as well as concrete examples of the issues presented.

# 6 Threats to Validity

A variety of errors is suspected in the analysis, model generation, and mapping of the selected DAW specification DSLs. Documentation from CWL, *Airflow*, *Nextflow*, *StackStorm*, and *Argo* provided information for the analysis. It was decided to analyse only this documentation and no other sources. Thus, only this documentation was relied upon. This could have prevented the development of valid models, as there was only a one-sided view of the facts.
It is likely that the interpretation of the information presented in the various language documentation is prone to error. Consequently, the derived language models could be flawed. As UML 2 class diagrams have been modeled, three main categories of diagram elements may contain errors. These elements are:

- Classes
- Relationships
- Multiplicities

As for the modeled classes and packages, they may have been misnamed. It may also be that there was a failure to include language elements in the form of classes. For some classes, there was discussion whether classes could be described as abstract or concrete classes. This was not always clear, and some room for error is suspected.
The relationships between different classes may have been chosen incorrectly and therefore do not correspond to the facts. When the models were created, it was not certain that all relationships between classes were modeled.
In addition, multiplicities attached to relationships in the models may describe incorrect values. In several cases, it was not sure whether a multiplicity starts at zero or one. A multiplicity that starts at zero would express that the associated class is optional and a workflow could be invoked without it.
Aside from the deficiencies in the language models, the mapping between language models and metamodels could also be flawed. The mapping of individual classes to generalized classes of the metamodel could be incorrect due to a possible misunderstanding of the concepts inherent in the languages and the metamodel.
Since this has occurred before, concerns can be raised about a possible discrepancy between the textual notations and the models created. Despite efforts to reconcile analysis and models, textual descriptions and diagrams may not match.

# 7 Future Work and Conclusion

Following this work, the language of DAWs was analyzed. At the beginning, basic concepts about DAWs, their design, mapping and implementation as well as the metamodel of Hilbrich et al. were introduced to create a clear understanding. Then, the steps of the applied methodology were mentioned. Five selected DAW specification DSLs were analyzed in light of the given metamodel. The elements inherent in each language were noted textually. Based on this, models were created in the form of UML 2 class diagrams. Mappings of the elements of each language model to the generalized elements of the metamodel were then performed. A brief evaluation of the language analysis performed was made. Reflections on the knowledge gained regarding the analysis approach were written down. In addition, the threats to validity of the analysis, the resulting models, and mappings were considered.
Based on these listed steps, future work can improve and expand this thesis. Although all models created were developed with care, future work should question model validity and review listed sources. Information about the selected languages from other sources could also be considered. In addition, uncertainties in language concepts should be addressed by developing test workflows that approach misinterpreted or poorly documented concepts.
The layout quality of the class diagrams presented could be optimized. The language models are currently constructed to provide largely planar mappings to the metamodel. However, other layouts are conceivable and may contribute to a better readability of the models and mappings.
The integration of visualizations in the form of further diagrams and concrete examples of workflows expressed by the respective languages could clarify the language concepts and provide for a broader understanding.
Since there are many more DAW specification DSLs, their analysis and mapping would be conceivable and could be seen as a direct continuation and extension of the thesis.
The analysis performed in the thesis serves to provide an overview of the DAW specification languages studied in regards to the metamodel. The mappings form a basis for comparability between the selected languages amidst the established metamodel. In the future, statements about the limitations and the power of the individual languages can be derived from the existing mappings. They also provide a basis for future translations between languages that have an existing mapping to the metamodel.
DAWs play an import role in the world of data-driven research as they guarantee reuse, reproducibility, and traceability of analysis results. Hence, advancements in the fields of DAWs and DAW specification languages are favorable. An increased portability across individual DAW concepts and platforms is a preferring subject to work on.
It could be demonstrated that a mapping from any of the selected languages to the metamodel is possible and that the basic principles inherent in the metamodel can be found in all the languages studied. In summary, the individual DAW specification DSLs correspond to the metamodel in [13] and thus answer the research questions posed.

## References

[1] Malcolm Atkinson, Sandra Gesing, Johan Montagnat, and Ian Taylor. “Scientific workflows: Past, present and future”. In: *Future Generation Computer Systems, Volume 75, Pages 216-227. ISSN 0167-739X* (2017).

[2] Chee Sun Liew, Malcolm P. Atkinson, Michelle Galea, Tan Fong Ang, Paul Martin, and Jano I. Van Hemert. “Scientific Workflows: Moving Across Paradigms”. In: *ACM Comput. Surv. 49, 4, Article 66 (December 2017), 39 pages* (2016).

[3] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. “A Survey of Data-Intensive Scientific Workflow Management”. In: *Grid Comput 13(4): 457–493* (2015).

[4] Nourhan Elfaramawy. “Interactive Workflows for Exploratory Data Analysis”. In: *Bao, Zhifeng; Sellis, Timos (Ed.): Proceedings of the VLDB 2022 PhD Workshopco-located with the 48th International Conference on Very Large Databases(VLDB2022), Sydney, Australia, September 5, 2022, CEUR-WS.org* (2022).

[5] Adam Barker and Jano van Hemert. “Scientific Workflow: A Survey and Research Directions”. In: *In: Roman Wyrzykowski, , Jack Dongarra, Konrad Karczewski, Jerzy Wasniewski (eds). Parallel Processing and Applied Mathematics. PPAM, 2007. Lecture Notes in Computer Science, vol 4967. Springer, Berlin, Heidelberg* (2008).

[6] Kota Miura and Nataša Sladoje. *Bioimage Data Analysis Workflows.* Springer Cham, 2020.

[7] Ulf Leser, Marcus Hilbrich, Claudia Draxl, Peter Eisert, Lars Grunske, Patrick Hostert, Dagmar Kainmüller, Odej Kao, Birte Kehr, Timo Kehrer, Christoph Koch, Volker Markl, Henning Meyerhenke, Tilmann Rabl, Alexander Reinefeld, Knut Reinert, Kerstin Ritter, Björn Scheuermann, Florian Schintke, Nicole Schweikardt, and Matthias Weidlich. “The Collaborative Research Center FONDA”. In: *Datenbank Spektrum 21, 255–260* (2021).

[8] Doris Dransch, Daniel Eggert, Nicola Abraham, Laurens M. Bouwer, Holger Brix, Ulrich Callies, Thomas Kalbacher, Stefan Lüdtke, Bruno Merz, Christine Nam, Erik Nixdorf, Daniela Rabe, Diana Rechid, Kai Schröter, Bente Tiedje, Dadiyorto Wendi, and Viktoria Wichert. “Data Analysis and Exploration with Scientific Workflows”. In: *In: Bouwer, L.M., Dransch, D., Ruhnke, R., Rechid, D., Frickenhaus, S., Greinert, J. (eds) Integrating Data Science and Earth Science. SpringerBriefs in Earth System Sciences. Springer, Cham* (2022).

[9] Sara Stoudt, Valeri N. Vasquez, and Ciera C. Martinez. “Principles for data analysis workflows”. In: *CoRR* abs/2007.08708 (2020).

[10] Christopher Schiefer, Marc Bux, Jörgen Brandt, Clemens Messerschmidt, Reinert Knut, Dieter Beile, and Ulf Leser. *Portability of Scientific Workflows in NGS Data Analysis: A Case Study.* 2020.

[11] Martin Fowler. *Domain-Specific Languages.* Pearson Education, 2010.

[12] Richard C. Gronback. *Eclipse Modeling Project: A DomainSpecific Language (DSL) Toolkit.* Pearson Education, 2009.

[13] Marcus Hilbrich, Sebastian Müller, Svetlana Kulagina, Christopher Lazik, Ninon De Mecquenem, and Lars Grunske. "A Consolidated View on Specification Languages for Data Analysis Workflows". In: *In Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering: 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22–30, 2022, Proceedings, Part II. Springer-Verlag, Berlin, Heidelberg, 201–215* (2022).

[14] Michael R. Crusoe, Sanne Abeln, Alexandru Iosup, Peter Amstutz, John Chilton, Nebojša Tijanić, Hervé Ménager, Stian Soiland-Reyes, Bogdan Gavrilovic, and Carole Gobleand. "Methods Included: Standardizing Computational Reuse and Portability with the Common Workflow Language". In: *Commun. ACM 65, 6 (June 2022), 54–63* (2021).

[15] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. *Metamodelling: State of the Art and Research Challenges.* 2014.

[16] CWL Project Team. *CWL user guide.* https://www.commonwl.org/user _guide/. Last accessed: 22.2.2023.

[17] The Apache Software Foundation. *What is Airflow?* https://airflow.apache.org/docs/apache-airflow/stable/. Last accessed: 22.2.2023.

[18] Seqera Labs. *Nextflow's documentation!* https://www.nextflow.io/docs/latest/index.html. Last accessed: 22.2.2023.

[19] LF Projects. *StackStorm 3.8.0 documentation.* https://docs.stackstorm.com/. Last accessed: 22.2.2023.

[20] Argo Project Authors. *Argo Workflows - The workflow engine for Kubernetes.* https://argoproj.github.io/argo-workflows/. Last accessed: 22.2.2023.

[21] Markus Völter, Sebastian Benz, Christian J. Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* 2013.

[22] Jason Agron. "Domain-Specific Language for HW/SW Co-design for FPGAs". In: *Taha, W.M. (eds) Domain-Specific Languages. DSL 2009. Lecture Notes in Computer Science, vol 5658. Springer, Berlin, Heidelberg* (2009).

[23] Diomidis D. Spinellis. "Notable design patterns for domain-specific languages". In: *Journal of Systems and Software. 56 : 91-99* (2001).

[24] Marc Bux and Ulf Leser. "Parallelization in Scientific Workflow Management Systems. Distributed, Parallel, and Cluster Computing (cs.DC)". In: *FOS: Computer and information sciences, FOS: Computer and information sciences, C.1.4; D.1.3; D.3.2; J.3, 68N19* (2013).

[25] Paolo Romano. “Automation of in-silico data analysis processes through workflow management systems”. In: *Briefings in Bioinformatics 9.1, 57-68* (2007).

[26] Nadia Cerezo, Johan Montagnat, and Mireille Blay-Fornarino. “Computer-Assisted Scientific Workflow Design”. In: *J Grid Computing 11, 585–612* (2013).

[27] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis, Gannon, Carole Goble, Miron Livny, Luc Moreau, and Jim Myers. “Examining the Challenges of Scientific Workflows”. In: *Computer. 40. 24 - 32. 10.1109/MC.2007.421* (2008).

[28] Bertram Ludäscher, Mathias Weske, Timothy McPhillips, and Shawn Bowers. “Scientific Workflows: Business as Usual?” In: *Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds) Business Process Management. BPM 2009. Lecture Notes in Computer Science, vol 5701. Springer, Berlin, Heidelberg* (2009).

[29] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2009.

[30] Manfred A. Jeusfeld. “Metamodel”. In: *LIU, L., ÖZSU, M.T. (eds) Encyclopedia of Database Systems. Springer, Boston* (2009).

[31] Chris Rupp, Stefan Queins, and Barbara Zengler. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. Hanser, 2007.

[32] Peter Fischer and Peter Hofer. *Lexikon der Informatik*. Springer Berlin, Heidelberg, 2010.

[33] Fazle Rabbi, Hao Wang, and Wendy MacCaull. “YAWL2DVE: An Automated Translator for Workflow Verification, 53-59”. In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*. 2010.

[34] Yuan Lin, Thérèse Libourel, and Isabelle Mougenot. “A Workflow Language for the Experimental Sciences”. In: 2009, pp. 372–375.

[35] CWL Project Team. *Common Workflow Language (CWL) Workflow Description, v1.2*. https://www.commonwl.org/v1.2/Workflow.html. Last accessed: 22.2.2023.

[36] CWL Project Team. *Common Workflow Language (CWL) Command Line Tool Description, v1.2*. https://www.commonwl.org/v1.2/CommandLineTool.html. Last accessed: 22.2.2023.

[37] LF Projects. *StackStorm Features*. https://stackstorm.com/features/. Last accessed: 22.2.2023.

[38] Peter Amstutz, Maxim Mikheev, Michael R. Crusoe, Nebojša Tijanić, Samuel Lampa, et al. *Existing Workflow systems. Common Workflow Language wiki, GitHub*. https://s.apache.org/existing-workflow-systems. Last accessed: 25.1.2023.

[39] Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay Fornarino. “A Data-Driven Workflow Language for Grids Based on Array Programming Principles”. In: WORKS ’09. Association for Computing Machinery, 2009.

# List of Figures

## Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 22ten Februar, 2023 ............ [illegible]