

# **SWEL: SCIENTIFIC WORKFLOW EXECUTION LANGUAGE**

*Rubén Salado-Cid and José Raúl Romero*

June 27, 2023  
Technical Report

# **SWEL: SCIENTIFIC WORKFLOW EXECUTION LANGUAGE**

*Rubén Salado-Cid and José Raúl Romero*

June 27, 2023

Technical Report

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Language rationale</b>	<b>2</b>
<b>3</b>	<b>Language structure</b>	<b>4</b>
<b>4</b>	<b>Morphological layer</b>	<b>5</b>
4.1	Package <i>ExecutionGraph</i> . . . . .	5
4.1.1	Rationale . . . . .	6
4.1.2	Metaclass <i>ExecutionGraph</i> . . . . .	7
4.1.3	Metaclass <i>Node</i> . . . . .	8
4.1.4	Metaclass <i>DirectedEdge</i> . . . . .	9
4.1.5	Metaclass <i>ControlLine</i> . . . . .	9
4.1.6	Metaclass <i>ExceptionLine</i> . . . . .	10
4.1.7	Metaclass <i>DataLine</i> . . . . .	11
4.1.8	Metaclass <i>Endpoint</i> . . . . .	11
4.1.9	Metaclass <i>ControlEndpoint</i> . . . . .	12
4.1.10	Metaclass <i>ExceptionEndpoint</i> . . . . .	13
4.1.11	Metaclass <i>DataEndpoint</i> . . . . .	13
4.1.12	Metaclass <i>Linker</i> . . . . .	14
4.1.13	Metaclass <i>LabelledElement</i> . . . . .	14
<b>5</b>	<b>Syntactic layer</b>	<b>15</b>
5.1	Package <i>Workflow</i> . . . . .	15
5.1.1	Rationale . . . . .	15
5.1.2	Metaclass <i>Workflow</i> . . . . .	16
5.1.3	Metaclass <i>DataDrivenWorkflow</i> . . . . .	18
5.1.4	Metaclass <i>ControlledDataDrivenWorkflow</i> . . . . .	18
5.1.5	Metaclass <i>ControlDrivenWorkflow</i> . . . . .	19
5.1.6	Metaclass <i>Constructor</i> . . . . .	20
5.1.7	Metaclass <i>ControlStructure</i> . . . . .	20
5.1.8	Metaclass <i>DataProvider</i> . . . . .	21
5.1.9	Metaclass <i>Activity</i> . . . . .	22
5.1.10	Metaclass <i>Port</i> . . . . .	22
5.1.11	Metaclass <i>ExceptionPath</i> . . . . .	23
5.1.12	Metaclass <i>Action</i> . . . . .	24
5.1.13	Metaclass <i>JumpTo</i> . . . . .	24
5.1.14	Metaclass <i>Checkpoint</i> . . . . .	25
5.1.15	Metaclass <i>Replicate</i> . . . . .	25
5.1.16	Metaclass <i>Retry</i> . . . . .	26
5.1.17	Metaclass <i>Abort</i> . . . . .	27
5.1.18	Metaclass <i>AnnotatedElement</i> . . . . .	27
5.2	Package <i>Activities</i> . . . . .	28
5.2.1	Rationale . . . . .	29
5.2.2	Metaclass <i>InteractionTask</i> . . . . .	29
5.2.3	Metaclass <i>ComputationalTask</i> . . . . .	30
5.2.4	Metaclass <i>DisplayTask</i> . . . . .	31
5.2.5	Metaclass <i>Service</i> . . . . .	31
5.2.6	Metaclass <i>WebService</i> . . . . .	32
5.2.7	Metaclass <i>RestClient</i> . . . . .	32
5.2.8	Metaclass <i>SoapClient</i> . . . . .	33

5.2.9	Metaclass <i>Process</i> . . . . .	34
5.2.10	Metaclass <i>JavaProcess</i> . . . . .	34
5.2.11	Metaclass <i>CliProcess</i> . . . . .	35
5.2.12	Metaclass <i>EmbeddedDisplay</i> . . . . .	36
5.2.13	Metaclass <i>DelegatedDisplay</i> . . . . .	36
5.3	Package <i>DataProviders</i> . . . . .	37
5.3.1	Rationale . . . . .	38
5.3.2	Metaclass <i>Record</i> . . . . .	38
5.3.3	Metaclass <i>Resource</i> . . . . .	38
5.3.4	Metaclass <i>File</i> . . . . .	39
5.3.5	Metaclass <i>Stream</i> . . . . .	40
5.3.6	Metaclass <i>Database</i> . . . . .	40
5.3.7	Metaclass <i>View</i> . . . . .	41
5.3.8	Enumeration <i>HttpMethod</i> . . . . .	42
5.4	Package <i>ControlStructures</i> . . . . .	42
5.4.1	Rationale . . . . .	43
5.4.2	Metaclass <i>Conditional</i> . . . . .	44
5.4.3	Metaclass <i>Begin</i> . . . . .	44
5.4.4	Metaclass <i>End</i> . . . . .	45
5.4.5	Metaclass <i>Fork</i> . . . . .	45
5.4.6	Metaclass <i>Synchronizer</i> . . . . .	46
5.4.7	Metaclass <i>Merge</i> . . . . .	47
5.4.8	Metaclass <i>Counter</i> . . . . .	47
5.4.9	Metaclass <i>Condition</i> . . . . .	48
5.4.10	Metaclass <i>Operand</i> . . . . .	49
5.4.11	Metaclass <i>Operator</i> . . . . .	49
5.4.12	Metaclass <i>LogicalOperator</i> . . . . .	50
5.4.13	Metaclass <i>RelationalOperator</i> . . . . .	51
5.4.14	Metaclass <i>MathematicalOperator</i> . . . . .	51
5.4.15	Metaclass <i>ANDOperator</i> . . . . .	52
5.4.16	Metaclass <i>OROperator</i> . . . . .	52
5.4.17	Metaclass <i>NOTOperator</i> . . . . .	53
5.4.18	Metaclass <i>XOROperator</i> . . . . .	54
5.4.19	Metaclass <i>GreaterOperator</i> . . . . .	54
5.4.20	Metaclass <i>GreaterEqualOperator</i> . . . . .	55
5.4.21	Metaclass <i>LessOperator</i> . . . . .	55
5.4.22	Metaclass <i>LessEqualOperator</i> . . . . .	56
5.4.23	Metaclass <i>EqualOperator</i> . . . . .	56
5.4.24	Metaclass <i>NotEqualOperator</i> . . . . .	57
5.4.25	Metaclass <i>SumOperator</i> . . . . .	58
5.4.26	Metaclass <i>SubtractionOperator</i> . . . . .	58
5.4.27	Metaclass <i>ProductOperator</i> . . . . .	59
5.4.28	Metaclass <i>DivisionOperator</i> . . . . .	59
5.4.29	Metaclass <i>ModuleOperator</i> . . . . .	60
5.5	Package <i>DataTypes</i> . . . . .	60
5.5.1	Rationale . . . . .	61
5.5.2	Metaclass <i>DataType</i> . . . . .	62
5.5.3	Metaclass <i>BasicDataType</i> . . . . .	62
5.5.4	Metaclass <i>ComplexDataType</i> . . . . .	63
5.5.5	Metaclass <i>Boolean</i> . . . . .	64
5.5.6	Metaclass <i>Integer</i> . . . . .	64
5.5.7	Metaclass <i>Real</i> . . . . .	65

5.5.8	Metaclass <i>String</i> . . . . .	65
5.5.9	Metaclass <i>RawText</i> . . . . .	66
5.5.10	Metaclass <i>Dataset</i> . . . . .	66
5.5.11	Metaclass <i>DOM</i> . . . . .	67
5.5.12	Metaclass <i>RawBinary</i> . . . . .	68
5.5.13	Metaclass <i>Picture</i> . . . . .	68
5.5.14	Metaclass <i>Audio</i> . . . . .	69
5.5.15	Metaclass <i>Video</i> . . . . .	69
5.5.16	Metaclass <i>Object</i> . . . . .	70
5.6	Package <i>ExceptionTypes</i> . . . . .	70
5.6.1	Rationale . . . . .	71
5.6.2	Metaclass <i>ExceptionType</i> . . . . .	71
5.6.3	Metaclass <i>BadParameterException</i> . . . . .	72
5.6.4	Metaclass <i>DiskFullException</i> . . . . .	73
5.6.5	Metaclass <i>PermissionDeniedException</i> . . . . .	73
5.6.6	Metaclass <i>ResourceUnreachableException</i> . . . . .	74
5.6.7	Metaclass <i>RuntimeException</i> . . . . .	74
5.6.8	Metaclass <i>TimeoutException</i> . . . . .	75
5.6.9	Metaclass <i>UnknownException</i> . . . . .	75
5.7	Package <i>ComputationalSpecification</i> . . . . .	76
5.7.1	Rationale . . . . .	77
5.7.2	Metaclass <i>ComputationalResource</i> . . . . .	77
5.7.3	Metaclass <i>Host</i> . . . . .	78
5.7.4	Metaclass <i>OperatingSystem</i> . . . . .	78
5.7.5	Metaclass <i>CPU</i> . . . . .	79
5.7.6	Metaclass <i>GPU</i> . . . . .	80
5.7.7	Enumeration <i>OSType</i> . . . . .	81
5.7.8	Enumeration <i>ArchitectureType</i> . . . . .	81
<b>6</b>	<b>Specification layer</b>	<b>82</b>
6.1	Package <i>WFSpecification</i> . . . . .	82
6.1.1	Rationale . . . . .	82
6.1.2	Metaclass <i>WFSpecification</i> . . . . .	83
6.1.3	Metaclass <i>Project</i> . . . . .	84
6.1.4	Metaclass <i>Stakeholder</i> . . . . .	85
6.1.5	Metaclass <i>Person</i> . . . . .	85
6.1.6	Metaclass <i>Organization</i> . . . . .	86
6.2	Package <i>ExperimentSpecification</i> . . . . .	86
6.2.1	Rationale . . . . .	87
6.2.2	Metaclass <i>Experiment</i> . . . . .	87
6.2.3	Metaclass <i>BasicExperiment</i> . . . . .	88
6.2.4	Metaclass <i>Hypothesis</i> . . . . .	89
6.2.5	Metaclass <i>Configuration</i> . . . . .	89
	<b>Appendices</b>	<b>91</b>
	<b>A Exemplary graphical and textual concrete syntax</b>	<b>91</b>
	<b>B Illustrative matching with existing concrete syntax</b>	<b>93</b>
	<b>C Workflow examples with SWEL</b>	<b>93</b>
C.1	Discover diseases . . . . .	95
C.2	Outlier detection in medical claims . . . . .	95



## List of Figures

1	Multi-layered structure of SWEL . . . . .	5
2	ExecutionGraph package . . . . .	6
3	Workflow package . . . . .	16
4	Activities package . . . . .	28
5	Data Providers package . . . . .	37
6	Control Structures package . . . . .	42
7	Data Types package . . . . .	61
8	Exception Types package . . . . .	71
9	Computational Specification package . . . . .	76
10	WFSpecification package . . . . .	83
11	Experiment Specification package . . . . .	87
12	SWEL representation of a data-driven workflow. . . . .	92
13	SWEL representation of a control-driven workflow. . . . .	92
14	Workflow using Taverna concrete syntax . . . . .	95
15	Workflow using LONI Pipeline concrete syntax . . . . .	96
16	Workflow using Taverna concrete syntax . . . . .	97
17	Workflow using LONI Pipeline concrete syntax . . . . .	97
18	SWEL simple workflow representation. . . . .	98

## List of Tables

1	Metaclasses of ExecutionGraph package . . . . .	7
2	Metaclasses of Workflow package . . . . .	17
3	Metaclasses of Activities package . . . . .	29
4	Data Providers package . . . . .	37
5	Control Structures package . . . . .	43
6	Data Types package . . . . .	61
7	Exception Types package . . . . .	71
8	Computational Specification package . . . . .	77
9	WFSpecification package . . . . .	82
10	Experiment Specification package . . . . .	87
11	Partial examples of concrete syntax. . . . .	91
12	Illustrative matching with partial Taverna concrete syntax . . . .	94
13	Illustrative matching with partial LONI Pipeline concrete syntax	94



# SWEL: Scientific Workflow Execution Language

## Technical Report

June 27, 2023

### **Abstract**

This technical report defines the formal specification of SWEL (*Scientific Workflow Execution Language*) in terms of its metamodels and constraints, making this definition syntax-independent, according to the precepts of model-driven engineering. SWEL is a domain-specific modelling language for defining data-intensive workflows (also known as *scientific workflows*), which takes advantage of computing techniques to perform computationally intensive executions. The objective of SWEL is to provide a high-level, domain-independent definition language for the formulation of data-intensive workflows, hiding low-level computational aspects. SWEL is founded on a multi-layered structure that enables both the definition of workflows for different domains, the reuse of knowledge and the portability and interoperability among different workflow platforms.

## **1 Introduction**

The amount of data collected by companies and organisations is growing exponentially, as well as their need to make the most of it by extracting useful and new knowledge. In this context, the so-called data-intensive (DI) applications aim at discovering valuable knowledge from huge amounts of data coming from real-world sources. These applications are becoming commonplace in many domains including, e.g. e-commerce, financial markets, manufacturing, marketing, education, and social sciences. The scientific sector is particularly interested in DI application because many research areas have become highly data-driven, such as bioinformatics, astronomy, or healthcare.

Regardless of the field of application, the knowledge discovery process of any DI application is normally formulated as a pipeline containing the sequence of individual tasks that must be completed to obtain meaningful and comprehensive results. These tasks typically include, among others, data acquisition, cleansing and preparation, information analysis, and data visualisation. Representing such pipelines as data-intensive workflows (DIWs), a.k.a. scientific workflows, enables the high-level definition of these processes and improves the understanding of the DI tasks by those professionals who are not skilled in data science, but are experts in their respective domains. In essence, DIWs provide bridges between data scientists, domain specialists, and the target computing infrastructure.

In this context, workflow technology is a suitable technology to define data-intensive workflows (or *scientific workflows*) at a high level of abstraction, without the need to know low-level computing details. Thus, a workflow collects and

defines knowledge as a multi-step computational procedure, such as retrieving data from an instrument or a database and changing the format of these data. The aim is to run a data analysis process to extract valuable information.

Some workflow languages have been already developed to define scientific workflows, such as SCUFL, MoML or CWL. Nevertheless, these languages have not been formally defined, but they are based on a given concrete syntax. Consequently, their applicability depends on specific implementations and tools, what drastically hampers interoperability and the — partial or total — reuse of knowledge, e.g. sharing workflows among different tools.

MDE (*Model-Driven Engineering*) addresses such difficulties, providing a set of tools that formally define the specification of DSMLs (*Domain-Specific Modelling Language*) in terms of models, metamodels and constraints. In MDE, a model is a simplified representation of a system or real-world concept, which is defined by a DSML. The abstract syntax of such DSML is a metamodel that explains and defines the relationships among the elements of the model. Thus, a language whose abstract syntax is defined by a metamodel improves and enables maintainability, reusability, portability and interoperability.

SWEL is a DSML providing domain experts — including data scientists — a precise, complete specification of the abstract syntax composed of all elements, relationships and constraints required to fully describe data-intensive workflows at both a high level of abstraction or a low level if any concrete syntax is declared. This language is based on a three-layer architecture, where each layer is focused on different aspects involved in the definition of a scientific workflow: from elements to define the underlying execution graph, elements that capture domain concepts, and to elements define human-readable information about scientific experiments.

The abstract syntax of SWEL and its semantics is given in the form of a metamodel that is independent of any concrete syntax, and consequently of any particular implementation or tool.

According to the structure and elements of SWEL, its main objectives are the following:

- To meet the needs to design and execute data-intensive applications at a high level of abstraction on any data-driven domain.
- To provide a platform-agnostic scientific workflow language.
- To enable the reuse and share of knowledge captured by scientific workflows.

Thus, SWEL can be used to reach interoperability between workflow languages, to enable the portability of knowledge, and to develop one or more consistent textual and graphical concrete syntaxes.

The rest of this document is organised as follows. The context, motivations and decisions taken for the design and development of SWEL are explained in Section 2. Then, Section 3 defines the multi-layered structure of the language, as well as the respective scope of each layer. Section 4, Section 5 and Section 6 describe in detail the language elements composing each specific layer.

## 2 Language rationale

The DSL must facilitate the creation of platform-agnostic and high-level DIWs that allow knowledge to be shared. With this aim, actions were taken to identify the basic elements common to DIWs, namely, the core elements of the DSL.

Initially, several meetings were conducted with different data scientists from industry and academia. These meetings allowed us to identify the most commonly used tools for DIW design, which were then supplemented with other similar tools found in the literature. During this search, we discovered that some of these tools were proprietary, and information about their DIW implementation details was not openly available. These solutions were not considered in the process, although their reference manuals were consulted when available for download. Next, we conducted a literature review on DIW languages used by these tools. Finally, we relied on existing studies on common characteristics, frequently used execution models, and the typical tasks required for creating a DIW.

As a result, we have identified the core elements of SWEL that enable the definition of a wide range of DIWs. These elements pertain to both the structure of the workflow and its specification.

**Workflow structure.** A workflow consists of interconnected tasks that define their dependencies. Its structure determines how these tasks and relationships are represented and executed. Generally, there are two types of representations: those based on DAG (Directed Acyclic Graph) and those based on DCG (Directed Cyclic Graph). Both have traditionally been used to define DIWs, although DAG-based representations are commonly used in data-driven domains. However, DCGs could be more appropriate for some business logic-driven domains. Therefore, SWEL must support both types to represent the widest range of DIWs possible and facilitate reuse across a greater number of tools.

**Workflow specification.** The definition of various types of tasks relies on the type of DIW language used. Abstract languages describe tasks at a high level of abstraction that does not reference specific platforms, computational resources, or programming languages. Such workflows are not directly executable but have to undergo a conversion stage that associates them with specific computational resources. The mapping responsibility usually falls on the execution engine or tool. In contrast, concrete languages contain specific tasks that take into account low-level implementation details, making them directly executable. Currently, there are more concrete languages than abstract ones since most languages have been developed during the creation of a WfMS like Taverna’s SCUFL or Kepler’s MoML. Nevertheless, some platform-independent abstract languages like CWL also exist.

SWEL should support both high-level abstraction tasks and more specific tasks associated with low-level aspects such as programming language, execution platform (cloud, grid, etc.), or execution models (sequential, parallel, etc.). Having high-level abstraction elements is essential for knowledge reuse between tools. However, since most tools use low-level definitions, it is necessary to provide elements that enable their definition, such as the invocation of web services or the execution of programs written in a specific programming language, among others.

After identifying the core elements, the next step is to identify the tasks involved in a DIW. To accomplish this, we conducted a search and review of various publicly available DIW languages. We used these languages to identify the primary computational and visualisation tasks, control structures, and commonly used input and output data providers.

Since not all languages have a model-based specification, we also performed a reverse engineering process by analysing their textual serialisation, which is usually based on XML and JSON. We created several DIWs in different tools with the support of specialists and used a large number of DIWs from the public DIW repository myExperiment.org.

Note that SWEL is designed to represent DIW at a high level of abstraction (Section 5). Then, to make it practicable, we have included more detailed elements that are designed to cover the most common tasks available for the most commonly used tools. The aim is that, while still being based on an extensible abstract language, SWEL can be augmented to incorporate new, more specific elements commonly used in workflow tools.

During the identification of high-level tasks, we discovered that the core elements identified in the previous phase were applicable to the analysed workflows. We also identified new elements that were not previously identified and incorporated them into SWEL as core elements. These new elements are related to the management of exceptions. When an exception breaks the normal flow of execution, it is necessary to execute a set of pre-defined exception handlers. Exception scenarios are required in some high-performance domains where the workflow languages have been designed to execute data-intensive workflows, in particular, underlying executing platforms, such as cloud or grids.

### 3 Language structure

The language is specified by a metamodel, whose elements enable the definition of scientific workflows. The different types of elements that form the metamodel are:

- Metaclasses: A metaclass is the basic unit that represents a particular concept of the language. Each metaclass is defined by:
  - Metaproperties: A set of properties that determine the behaviour of the metaclass in the language. The properties that define a metaclass are: *isAbstract* to indicate that the metaclass cannot be directly used in the language, *isFinal* to indicate that the metaclass cannot be specialised and, finally, *isExtensible* to determine if the metaclass can be extended to be used in a particular domain.
  - Generalisation: The list of metaclasses of which the metaclass inherits attributes and relationships.
  - Specialisation: The list of metaclasses that inherit the attributes and relationships of the metaclass.
  - Attributes: The set of values that define the metaclass.
  - Associations: The list of relationships that the metaclass has with other metaclasses.
  - Constraints: The set of rules that restrict the behaviour of the metaclass in the language.
- Packages: A package is a grouping of metaclasses that represent similar or related concepts. A package can import the metaclasses from other packages to extend or modify its definition.
- Layer: A layer sets a theoretical division between packages, which are defined at a different level of abstraction.

SWEL exhibits a 3-layered architecture (Figure 1), where each layer is focused on different aspects of the language definition.

- Morphological layer: This layer contains the basic low-level elements to define an executable graph as a set of nodes and connections. These elements are extended in the next layer.
- Syntactic layer: This layer extends the elements provided by the Morphological Layer to add new elements. These new elements define a scientific workflow as a set of interconnected constructors that capture domain-specific requirements and resources.
- Specification layer: This layer adds new elements to extend the capabilities of the Syntactic Layer, providing elements that gather human-readable information related to the computationally intensive experiments defined by the scientific workflow.

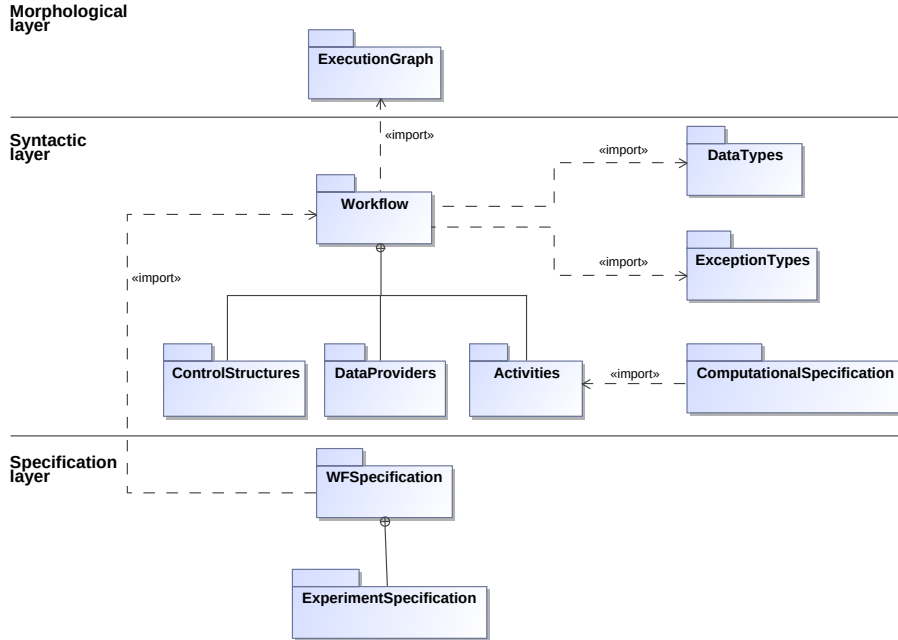


Figure 1: Multi-layered structure of SWEL

## 4 Morphological layer

This layer groups the packages intended to structure a scientific workflow as an executable, or computational, graph, which is composed of nodes and connections. The morphological layer is composed of a single package, *ExecutionGraph*.

### 4.1 Package *ExecutionGraph*

This package provides metaclasses to define executable graphs (Figure 2). Each node of this type of graph represents a computational task to be executed and the dependencies between tasks are expressed by connections. The metaclasses that compose this package are listed in Table 1.

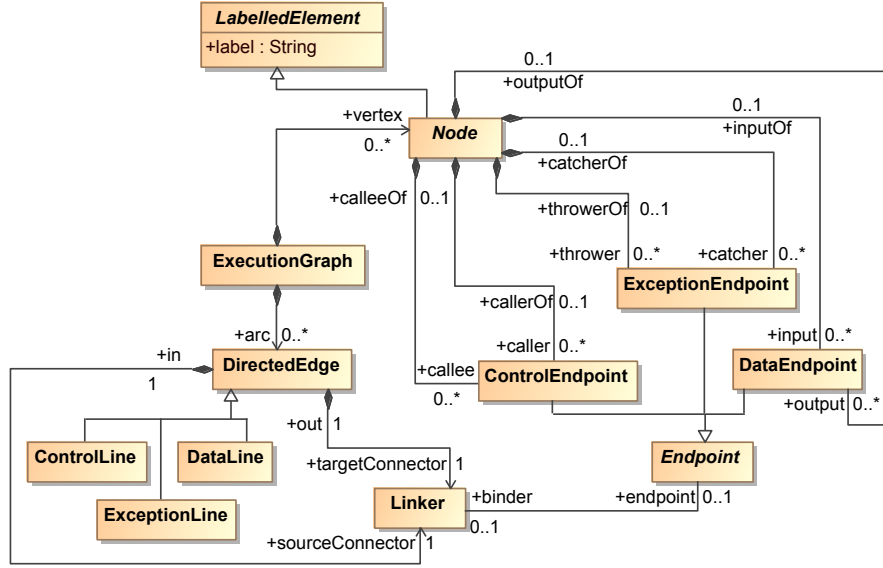


Figure 2: ExecutionGraph package

#### 4.1.1 Rationale

Upon analyzing a significant number of existing workflow definitions, a set of core elements has been identified. Though these elements are not explicitly represented in most languages, implicitly, all existing workflows are organized as a graph (*ExecutionGraph*) with a set of nodes (*Node*) and directed edges (*DirectedEdge*) between them. Thus, SWEL workflows are based on these elements to define the workflow structure, gathering the basic knowledge that indicates how the execution order is determined.

Specific elements have been identified to enable the definition of control flow and data flow requirements necessary to execute a workflow. These types of flows are required to define the different types of dependencies between the workflow elements. The SWEL control flow elements (*ControlLine* and *ControlEndpoint*) enable the definition of the temporal dependencies between nodes. This information is primarily used by business logic-oriented workflows. The SWEL data flow elements (*DataLine* and *DataEndpoint*) enable the definition of the data dependencies between nodes. In data-driven domains, data flow is the most crucial flow since the execution order is implicitly derived from data dependencies.

To model explicit relationships between nodes, it is necessary to represent directed edges and endpoints (*Endpoint*). Endpoints determine the entry and exit points of the node, while directed edges define which particular nodes are connected to each endpoint. These two conditions are essential to determine the execution flow of the workflow. The type of a directed edge can be implicitly derived from the connected endpoints. For example, if a directed edge connects two control endpoints, the directed edge will be a control line. If a directed edge links two data endpoints, the directed edge will be a data line. In contrast, a directed edge linking different types of endpoints will not be a valid directed edge.

It should be noted that some tools, such as Taverna, consider different types

Table 1: Metaclasses of ExecutionGraph package

Metaclass	Description	Section
ExecutionGraph	Graph composed by executable nodes and directed edges	4.1.2
Node	Execution unit of a execution graph	4.1.3
DirectedEdge	Connection between nodes of an execution graph	4.1.4
ControlLine	Control dependency between nodes	4.1.5
ExceptionLine	Dependency between nodes when an error is produced	4.1.6
DataLine	Data dependency between nodes	4.1.7
Endpoint	Connection point of a node	4.1.8
ControlEndpoint	Connection point for control lines	4.1.9
ExceptionEndpoint	Connection point for exception lines	4.1.10
DataEndpoint	Connection point for data lines	4.1.11
Linker	Relationship between a directed edge and an endpoint	4.1.12
LabelledElement	Element with name	4.1.13

of directed edges to provide a specific representation, such as a dotted line for data lines and a solid line for control lines. In this scenario, it is necessary to provide different metaclasses to each type.

#### 4.1.2 Metaclass *ExecutionGraph*

An execution graph specifies the flow of information in a multi-step computational process. This graph is composed by a set of vertices or nodes, which can be connected by arcs or edges.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

##### *Generalisation*

N/A

##### *Specialisation*

N/A

##### *Attributes*

N/A

##### *Associations*

- arc : DirectedEdge [0..\*]  
Set of arcs used to connect nodes

- vertex : Node [0..\*]  
Set of vertices that specify the different steps of the execution graph

### ***Constraints***

N/A

#### **4.1.3 Metaclass *Node***

A node defines the fundamental unit of execution of which execution graphs are composed. A node can be connected to other nodes by means of its endpoints or connection points, or exists in an execution graph and not be connected.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- LabelledElement (section 4.1.13)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- output : DataEndpoint [0..\*]  
Set of connection points to specify data output from the node
- caller : ControlEndpoint [0..\*]  
Set of connection points to specify the next node to execute
- callee : ControlEndpoint [0..\*]  
Set of connection points to specify that the node has to be executed after a previous node
- input : DataEndpoint [0..\*]  
Set of connection points to specify data input to the node
- thrower : ExceptionEndpoint [0..\*]  
Set of connection points to specify that an error situation has been produced during node execution
- catcher : ExceptionEndpoint [0..\*]  
Set of connection points to manage an error situation that has been propagated from a previous node



### ***Constraints***

N/A

#### **4.1.4 Metaclass *DirectedEdge***

A directed edge defines a basic unit of which execution graphs are constructed. A directed edge specifies the connection between a source node and a target node in the execution graph, which is defined by a linker.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

- ControlLine (section 4.1.5)
- ExceptionLine (section 4.1.6)
- DataLine (section 4.1.7)

### ***Attributes***

N/A

### ***Associations***

- targetConnector : Linker [1]  
The linker to specify the target node.
- sourceConnector : Linker [1]  
The linker to specify the source node.

### ***Constraints***

- “targetConnector” and “sourceConnector” must be different.

#### **4.1.5 Metaclass *ControlLine***

A control line is a type of directed edge that specifies the control dependency existing between nodes, defining that a node has to be executed after another node.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- DirectedEdge (section 4.1.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

- “sourceConnector” can only be associated with a linker whose endpoint is ControlEndpoint.
- “targetConnector” can only be associated with a linker whose endpoint is ControlEndpoint.

#### **4.1.6 Metaclass *ExceptionLine***

An exception line is a type of directed edge that specifies the propagation of an error situation produced during the execution of a node.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- DirectedEdge (section 4.1.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

- “sourceConnector” can only be associated with a linker whose endpoint is ExceptionEndpoint.
- “targetConnector” can only be associated with a linker whose endpoint is ExceptionEndpoint.

#### **4.1.7 Metaclass *DataLine***

A data line specifies the flow of data between nodes.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- DirectedEdge (section 4.1.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

- “sourceConnector” can only be associated with a linker whose endpoint is DataEndpoint.
- “targetConnector” can only be associated with a linker whose endpoint is DataEndpoint.

#### **4.1.8 Metaclass *Endpoint***

An endpoint is a connection point of a node that enables the communication between nodes by means of directed edges. The type of endpoint determines the type of communication that can be established.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

- ControlEndpoint (section 4.1.9)
- ExceptionEndpoint (section 4.1.10)
- DataEndpoint (section 4.1.11)

### ***Attributes***

N/A

### ***Associations***

- binder : Linker [0..1]  
Linker used to specify the directed edge used to be bound to a node endpoint.

### ***Constraints***

N/A

#### **4.1.9 Metaclass *ControlEndpoint***

A control endpoint enables the connection of control lines to specify control dependency between nodes.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- Endpoint (section 4.1.8)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- callerOf : Node [0..1]  
Node where the endpoint enables calling a node to execute.
- calleeOf : Node [0..1]  
Node where the endpoint is used to be called by a node.

### ***Constraints***

- "callerOf" and "calleeOf" must be the same when both exist.

#### **4.1.10 Metaclass *ExceptionEndpoint***

An exception endpoint enables the connection of exception lines to specify error situations during execution of a node.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- Endpoint (section 4.1.8)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- catcherOf : Node [0..1]  
Node where the endpoint enables catching propagated errors.
- throwerOf : Node [0..1]  
Node where the endpoint enables propagating errors.

### ***Constraints***

- "catcherOf" and "throwerOf" must be the same when both exist.

#### **4.1.11 Metaclass *DataEndpoint***

A data endpoint enables the connection of data lines to specify the flow of data between nodes.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- Endpoint (section 4.1.8)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- **inputOf** : Node [0..1]  
Node where the endpoint enables the data input.
- **outputOf** : Node [0..1]  
Node where the endpoint enables the data output.

### ***Constraints***

- "inputOf" and "outputOf" must be the same when both exist.

#### **4.1.12 Metaclass *Linker***

A linker specifies the connection of a directed edge with a particular endpoint of a node.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- **endpoint** : Endpoint [0..1]  
Endpoint used to connect the node and the directed edge

### ***Constraints***

N/A

#### **4.1.13 Metaclass *LabelledElement***

A named element is an element that have a name. This allows that an element can be referenced using its name as identifier.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

- Node (section 4.1.3)

### ***Attributes***

- label : String [1]  
The element label

### ***Associations***

N/A

### ***Constraints***

N/A

## **5 Syntactic layer**

This layer groups the packages that capture the domain-specific requirements and resources in a workflow. The package *Workflow* provide metaclasses to define domain-specific tasks that are arranged as workflows, the package *DataTypes* contains metaclasses that specify the supported data types, the package *ExceptionTypes* provides the fault-tolerance handlers and, finally, *ComputationalSpecification* gathers metaclasses to define the computational resources required to execute the scientific workflow.

### **5.1 Package *Workflow***

This package provides metaclasses to define different types of scientific workflows (Figure 3). A scientific workflow is composed by a set of elements involved in the execution of a computational process, such as control structures, data providers and activities, and the flow of information between them. Such elements are listed in Table 2.

#### **5.1.1 Rationale**

A scientific workflow (*Workflow*) collects knowledge from data-intensive domains to perform a computational process. Therefore, this type of workflow includes the definition of resources involved in computing.

To abstract data scientists from low-level computing requirements, scientific workflows provide a set of high-level elements (*Constructor*) to define *what* to do, not *how* to do it. These elements are related to the reading and writing of

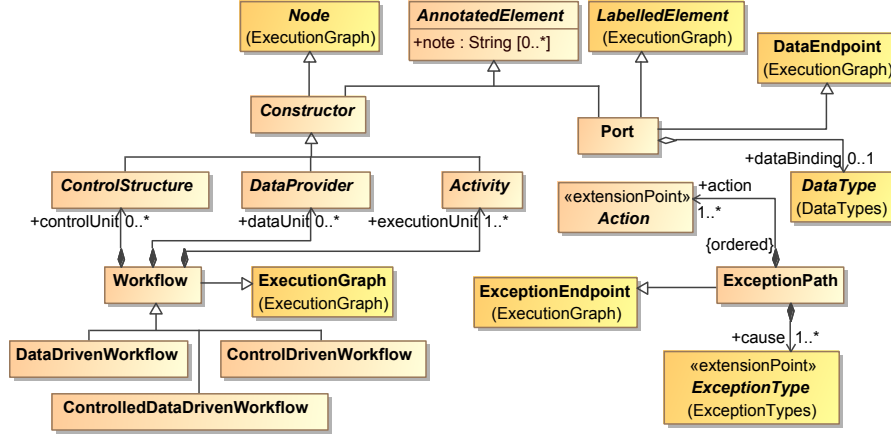


Figure 3: Workflow package

data (*DataProvider*), the execution of specific computing programs (*Activities*) or the control of execution order (*ControlStructure*).

Depending on the specific domain requirements, we have considered different types of workflows. This differentiation is important to support some particular existing features, such as the validation of the workflow definition to reduce human errors in the design step, such as adding a control structure to a data-driven workflow. Nevertheless, some tools delegate the validity checking to the underlying execution platform.

Regardless of the workflow type, data flows are a central asset in data-intensive computing (*Port*) in order to define the type of data accepted by the program. In this data-intensive scenario, we have identified that some tools are highly coupled to the underlying computing platform. Therefore, it is essential to enable the definition of exception paths (*ExceptionPath*) to handle executing errors at runtime.

### 5.1.2 Metaclass *Workflow*

A workflow specifies the sequence of steps and resources involved in an executable computational process, together with the flow of information required to coordinate and perform such steps. A workflow is arranged as a graph, where the steps are defined by nodes, and the flows of information are specified by directed edges.

Depending on the flows of information and steps supported, there are different types of workflows.

#### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

#### *Generalisation*

- ExecutionGraph::ExecutionGraph (section 4.1.2)



Table 2: Metaclasses of Workflow package

Metaclass	Description	Section
Workflow	Set of tasks and resources of a computational process	5.1.2
DataDrivenWorkflow	Workflow whose execution flow is only determined by data dependencies	5.1.3
ControlledDataDrivenWorkflow	Workflow whose execution flow is determined by data dependencies and control dependencies	5.1.4
ControlDrivenWorkflow	Workflow whose execution flow is only determined by control dependencies	5.1.5
Constructor	Step of a workflow	5.1.6
ControlStructure	Control unit of a workflow	5.1.7
DataProvider	Data unit of a workflow	5.1.8
Activity	Execution unit of a workflow	5.1.9
Port	Data types exposed by constructors	5.1.10
ExceptionPath	Alternative path when errors are produced	5.1.11
Action	Operation to perform when an error is produced	5.1.12
JumpTo	Next constructor to execute when an error is produced	5.1.13
Checkpoint	Save the current state of the execution	5.1.14
Replicate	Try the execution in a different host	5.1.15
Retry	Try the execution a particular number of times	5.1.16
Abort	Stop the workflow execution	5.1.17
AnnotatedElement	Elements with human-readable notes	5.1.18

***Specialisation***

- DataDrivenWorkflow (section 5.1.3)
- ControlledDataDrivenWorkflow (section 5.1.4)
- ControlDrivenWorkflow (section 5.1.5)

***Attributes***

N/A

***Associations***

- executionUnit : Activity [1..\*]  
Set of computational unit involved in the workflow.

- controlUnit : ControlStructure [0..\*]  
Set of control unit intended to manage the flow of execution of the workflow.
- dataUnit : DataProvider [0..\*]  
Set of data resources involved in the workflow.
- description : WFSpecification::WFSpecification [0..\*]  
Information about the project that led the design of the workflow.

### ***Constraints***

N/A

#### **5.1.3 Metaclass *DataDrivenWorkflow***

A data driven workflow is a workflow whose flow of execution is only controlled by the data dependencies.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Workflow (section 5.1.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

- “controlUnit” must be zero.

#### **5.1.4 Metaclass *ControlledDataDrivenWorkflow***

A controlled data driven workflow is a workflow whose flow of execution is mainly controlled by the data dependencies. However, the flow of execution can be defined by control dependencies when there are no data dependencies between nodes.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Workflow (section 5.1.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.5 Metaclass *ControlDrivenWorkflow***

A control driven workflow is a workflow whose flow of execution is only controlled by the control dependencies. A control dependency specifies the temporal order in which the nodes are executed.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Workflow (section 5.1.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.6 Metaclass *Constructor***

A constructor is the basic unit that composes a workflow. The set of constructors defines the sequence of steps and resources required by the workflow execution.

Depending on the type of steps or resources, there are different types of constructors.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ExecutionGraph::Node (section 4.1.3)
- AnnotatedElement (section 5.1.18)

### ***Specialisation***

- ControlStructure (section 5.1.7)
- DataProvider (section 5.1.8)
- Activity (section 5.1.9)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.7 Metaclass *ControlStructure***

A control structure is a control unit that defines the management of the workflow execution.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Constructor (section 5.1.6)

### ***Specialisation***

- Workflow::ControlStructures::Conditional (section 5.4.2)
- Workflow::ControlStructures::Begin (section 5.4.3)
- Workflow::ControlStructures::End (section 5.4.4)
- Workflow::ControlStructures::Fork (section 5.4.5)
- Workflow::ControlStructures::Synchronizer (section 5.4.6)
- Workflow::ControlStructures::Merge (section 5.4.7)
- Workflow::ControlStructures::Counter (section 5.4.8)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.1.8 Metaclass *DataProvider***

A data provider is a data unit that defines the data resources involved in a workflow. A data resource can be both consumed and produced by the different workflow constructors.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Constructor (section 5.1.6)

### ***Specialisation***

- Workflow::DataProviders::Record (section 5.3.2)
- Workflow::DataProviders::Resource (section 5.3.3)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.9 Metaclass *Activity***

An activity is a execution unit that defines a computational process to be executed.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Constructor (section 5.1.6)

### ***Specialisation***

- Workflow::Activities::InteractionTask (section 5.2.2)
- Workflow::Activities::ComputationalTask (section 5.2.3)
- Workflow::Activities::DisplayTask (section 5.2.4)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.10 Metaclass *Port***

A port specifies the type of data that can be consumed and produced by constructors.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ExecutionGraph::LabelledElement (section 4.1.13)
- ExecutionGraph::DataEndpoint (section 4.1.11)
- AnnotatedElement (section 5.1.18)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- dataBinding : DataTypes::DataType [0..1]  
The type of data that can be managed by the port.

### ***Constraints***

N/A

#### **5.1.11 Metaclass *ExceptionPath***

An exception path specifies the set of actions to perform when a particular error is produced.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ExecutionGraph::ExceptionEndpoint (section 4.1.10)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- cause : ExceptionTypes::ExceptionType [1..\*]  
List of causes that have produced an error.
- action : Action [1..\*]  
Ordered list of actions to perform when an error is produced.

### ***Constraints***

N/A

#### **5.1.12 Metaclass *Action***

An action specifies how the flow of execution have to be modified when an error is produced.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

### ***Generalisation***

N/A

### ***Specialisation***

- JumpTo (section 5.1.13)
- Checkpoint (section 5.1.14)
- Replicate (section 5.1.15)
- Retry (section 5.1.16)
- Abort (section 5.1.17)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.13 Metaclass *JumpTo***

This metaclass defines the constructor that have to be executed when an error is produced.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No



### ***Generalisation***

- Action (section 5.1.12)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- target : Constructor [1]  
The target constructor to execute when an error is produced.

### ***Constraints***

N/A

#### **5.1.14 Metaclass *Checkpoint***

This metaclass specifies that a checkpoint have to be created to save the state of the workflow execution.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Action (section 5.1.12)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.15 Metaclass *Replicate***

This metaclass specifies that the workflow constructor has to be executed in a different machine.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Action (section 5.1.12)

### ***Specialisation***

N/A

### ***Attributes***

- host : String [1]  
The host where to retry the execution of the workflow constructor.

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.1.16 Metaclass *Retry***

This metaclass defines that a workflow constructor has to be executed a particular number of times when an error is produced.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Action (section 5.1.12)

### ***Specialisation***

N/A

### ***Attributes***

- times : int [1]  
the number of times the execution should be retry.
- delay : int [1]  
the interval of time, in milliseconds, that has to elapse between each attempt.

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.17 Metaclass *Abort***

This metaclass determines that the workflow execution is cancelled when an error is produced.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Action (section 5.1.12)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.1.18 Metaclass *AnnotatedElement***

An annotated element is an element that have a set of information intended to be read by humans.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### Specialisation

- Constructor (section 5.1.6)
- Port (section 5.1.10)

### Attributes

- note : String [0..\*]  
List of information about the element.

### Associations

N/A

### Constraints

N/A

## 5.2 Package Activities

This package provides metaclasses that define different types of computational tasks (Figure 4). A computational task receives a set of input data that are processed to produce new output data. In Table 3 are listed all metaclasses.

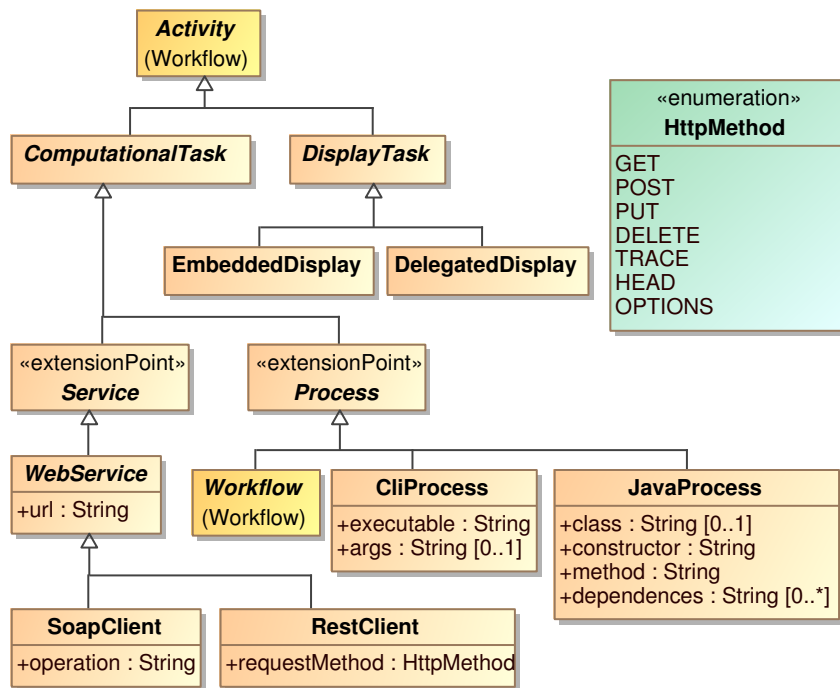


Figure 4: Activities package

Table 3: Metaclasses of Activities package

Metaclass	Description	Section
InteractionTask	Task that needs human intervention	5.2.2
ComputationalTask	Task executed by computers	5.2.3
DisplayTask	Task to show data information	5.2.4
Service	Service to be consumed by clients	5.2.5
WebService	Service served using HTTP protocol	5.2.6
RestClient	Client to consume REST services	5.2.7
SoapClient	Client to consume SOAP services	5.2.8
Process	Executable computing process	5.2.9
JavaProcess	Process to be executed using Java	5.2.10
CliProcess	Process that exposes a command-line interface	5.2.11
EmbeddedDisplay	Display to be show in the workflow	5.2.12
DelegatedDisplay	Display to be managed by an external tool	5.2.13
HttpMethod	Enumeration of HTTP methods	5.3.8

### 5.2.1 Rationale

The category of tasks is closely related to the workflow tool, which led to the development of the corresponding workflow language, allowing for many specific types of tasks, such as computational tasks (*ComputationalTask*) and display tasks (*DisplayTask*), to be used in defining workflows.

We have observed that existing workflow tools mainly use Java as the programming language, which leads to the prevalence of computational tasks written in Java (*JavaProcess*). Additionally, web services (*WebService*) and CLI programs (*CliProcess*) are commonly used in a number of DIWs. Some workflow languages even permit pre-existing workflows to be used as computational tasks (nested workflows), providing an interesting mechanism to increase the level of abstraction in defining new workflows.

As technology continually evolves, we have discovered that activities written in other programming languages, such as JavaScript or Python, or GraphQL web services, are being considered by some specific workflow languages. Therefore, we define extension points to enable SWEL users to extend SWEL, including new relevant services or processes for their use cases.

Besides computational tasks, display tasks are relevant in some specific workflow tools like VisTrails or Kepler. As data visualisation is an essential part of the data mining process, we have included particular elements to display data in the workflow or to delegate the visualisation to an external tool.

### 5.2.2 Metaclass *InteractionTask*

An interaction task is a task that required the involvement of a human to be performed.

#### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

### ***Generalisation***

- Workflow::Activity (section 5.1.9)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.2.3 Metaclass *ComputationalTask***

A computational task is a task that is automatically performed by a computer.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

### ***Generalisation***

- Workflow::Activity (section 5.1.9)

### ***Specialisation***

- Service (section 5.2.5)
- Process (section 5.2.9)

### ***Attributes***

N/A

### ***Associations***

- computationalDescription : ComputationalSpecification::ComputationalResource  
[0..\*]  
Specification of the computational resources required to be properly executed.

### ***Constraints***

N/A

#### 5.2.4 Metaclass *DisplayTask*

A display task is a task intended to only show information. This information can be displayed using tables, charts, images, etc.

##### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

##### *Generalisation*

- Workflow::Activity (section 5.1.9)

##### *Specialisation*

- EmbeddedDisplay (section 5.2.12)
- DelegatedDisplay (section 5.2.13)

##### *Attributes*

N/A

##### *Associations*

N/A

##### *Constraints*

N/A

#### 5.2.5 Metaclass *Service*

A service is a type of computational task that is running in any location. A service is typically consumed by clients.

##### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

##### *Generalisation*

- ComputationalTask (section 5.2.3)

##### *Specialisation*

- WebService (section 5.2.6)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.2.6 Metaclass *WebService***

A web service is a type of service that uses the HTTP protocol to be exposed and consumed by web clients.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Service (section 5.2.5)

### ***Specialisation***

- RestClient (section 5.2.7)
- SoapClient (section 5.2.8)

### ***Attributes***

- url : String [1]  
The URL of the web service

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.2.7 Metaclass *RestClient***

A REST client defines the consume of a web service that is directly exposed by a HTTP server.



### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- WebService (section 5.2.6)

### ***Specialisation***

N/A

### ***Attributes***

- requestMethod : HttpMethod [1]  
The HTTP method to use

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.2.8 Metaclass *SoapClient***

A SOAP client defines the consume of a web service that is exposed using the SOAP/WSDL protocol.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- WebService (section 5.2.6)

### ***Specialisation***

N/A

### ***Attributes***

- operation : String [1]  
The method to be consumed by the client.

### ***Associations***

N/A

### ***Constraints***

N/A

### **5.2.9 Metaclass *Process***

A process is an instance of a computer program that is executed to generate new data output.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

### ***Generalisation***

- ComputationalTask (section 5.2.3)

### ***Specialisation***

- JavaProcess (section 5.2.10)
- CliProcess (section 5.2.11)
- Workflow (section 5.1.2)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

### **5.2.10 Metaclass *JavaProcess***

A Java process is a type of process that defines the execution of a Java program.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Process (section 5.2.9)

### ***Specialisation***

N/A

### ***Attributes***

- class : String [0..1]  
The Java class that contains the method to execute.
- constructor : String [1]  
The Java constructor used to create the corresponding Java object.
- method : String [1]  
The Java method to execute.
- dependencies : String [0..\*]  
The Java dependencies required to execute the Java program.

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.2.11 Metaclass *CliProcess***

A command-line interface process is a type of process that defines the execution of a program that exposes a command-line interface.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Process (section 5.2.9)

### ***Specialisation***

N/A

### ***Attributes***

- executable : String [1]  
The path to the executable program.
- args : String [0..1]  
The arguments provided to the program when it is started.

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.2.12 Metaclass *EmbeddedDisplay***

An embedded display defines the configuration required for a graphical visualizer can show information on the workflow.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- DisplayTask (section 5.2.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.2.13 Metaclass *DelegatedDisplay***

A delegated display defines the configuration required for a external tool can show information managed by workflow.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- DisplayTask (section 5.2.4)

### ***Specialisation***

N/A

### *Attributes*

N/A

### *Associations*

N/A

### *Constraints*

N/A

## 5.3 Package *DataProviders*

This package provides metaclasses to define different type of data providers (Figure 5). A data provider is intended to manage the data types involved in a scientific workflow according to its origin and destination. In Table 4 are listed the corresponding metaclasses.

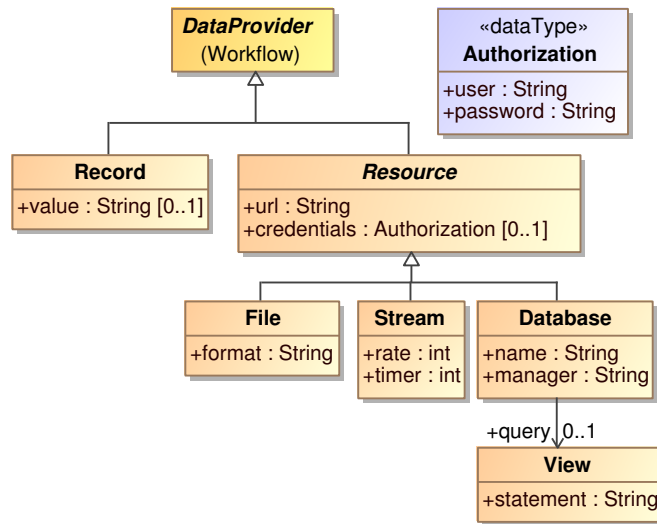


Figure 5: Data Providers package

Table 4: Data Providers package

Metaclass	Description	Section
Record	Data stored in main memory	5.3.2
Resource	Data stored in secondary memory	5.3.3
File	Data stored in a file	5.3.4
Stream	Data to be consumed as a data flow	5.3.5
Database	Data stored in a database	5.3.6
View	Subset of data obtained from a database	5.3.7

### 5.3.1 Rationale

A DIW takes input data and produces new output data. In order to store data, existing workflows use both main memory (*Record*) and secondary memory (*Resource*), with files (*File*) and databases (*Database*) being the most commonly used data providers. Some DIWs focus on processing data streams (*Stream*), which are typically managed by workflow engines as data batches. However, some workflow tools have been developed to process data streams properly.

We have observed that the range of available data providers in some tools is determined by the domain that led to the development of the tool. SWEL offers the most commonly used and widely compatible data providers, but this does not imply the ability to define SWEL workflows in domains with specific data providers. It is common for this responsibility to be delegated to computational tasks that allow for the integration of specific data providers beyond the basic ones providers.

### 5.3.2 Metaclass *Record*

A record defines a basic data structure that contains information stored in the main memory in a computer.

#### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

#### *Generalisation*

- Workflow::DataProvider (section 5.1.8)

#### *Specialisation*

N/A

#### *Attributes*

- value : String [0..1]  
The information stored in the main memory.

#### *Associations*

N/A

#### *Constraints*

N/A

### 5.3.3 Metaclass *Resource*

A resources defines information that is stored in a computer system, being accessible through a specific URL.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- DataProvider (section 5.1.8)

### ***Specialisation***

- File (section 5.3.4)
- Stream (section 5.3.5)
- Database (section 5.3.6)

### ***Attributes***

- url : String [1]  
The URL to access the information.
- credentials : Authorization [0..1]  
Authorization data to access the information.

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.3.4 Metaclass *File***

This metaclass defines the access to information stored in a file.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Resource (section 5.3.3)

### ***Specialisation***

N/A

### ***Attributes***

- format : String [1]  
The file format that contains the information.

### ***Associations***

N/A

### ***Constraints***

N/A

### **5.3.5 Metaclass *Stream***

This metaclass defines the access to information that have to be process while it is being transmitted.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Resource (section 5.3.3)

### ***Specialisation***

N/A

### ***Attributes***

- rate : int [1]  
The velocity of reading.
- timer : int [1]  
The amount of time used to know when a stream is not transmitting data.

### ***Associations***

N/A

### ***Constraints***

N/A

### **5.3.6 Metaclass *Database***

This metaclass defined the access to information stored in a database and that have to be accessed through database managment system.



### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Resource (section 5.3.3)

### ***Specialisation***

N/A

### ***Attributes***

- manager : String [1]  
URL to access the database.

### ***Associations***

- query : View [0..1]  
SQL statement to query data in the database.

### ***Constraints***

N/A

## **5.3.7 Metaclass *View***

A view defines a SQL statement to retrieve a set of data stores in a database.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

N/A

### ***Attributes***

- statement : String [1]  
The SQL statement to query data.

### ***Associations***

N/A

### Constraints

N/A

#### 5.3.8 Enumeration *HttpMethod*

HttpMethod is an enumeration type that specifies the literals for defining the HTTP method to be used.

### Values

- GET  
Indicates that the HTTP method is GET.
- POST  
Indicates that the HTTP method is POST.
- PUT  
Indicates that the HTTP method is PUT.
- DELETE  
Indicates that the HTTP method is DELETE.
- TRACE  
Indicates that the HTTP method is TRACE.
- HEAD  
Indicates that the HTTP method is HEAD.
- OPTIONS  
Indicates that the HTTP method is OPTIONS.

## 5.4 Package *ControlStructures*

This package provides metaclasses to define different types of control structures that manage the execution flow of the workflow (Figure 6). All control structures supported by SWEL are listed in Table 5.

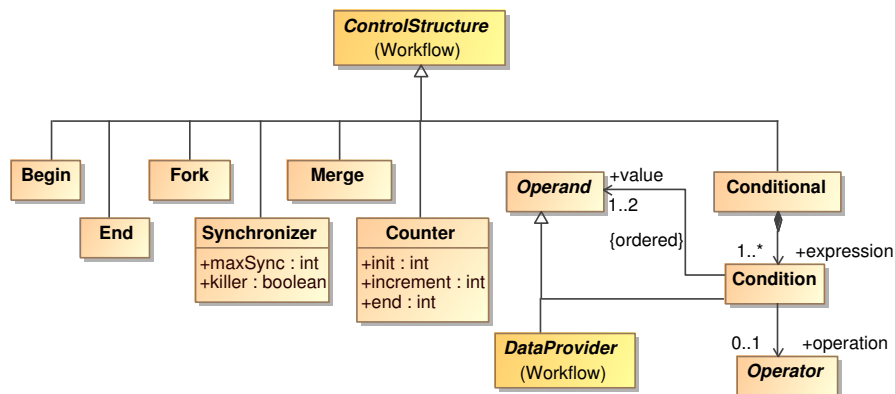


Figure 6: Control Structures package

Table 5: Control Structures package

Metaclass	Description	Section
Conditional	Determine the next constructor to execute based on a condition	5.4.2
Begin	Determines the first constructors to execute	5.4.3
End	Determine the last constructors to execute	5.4.4
Fork	Determine the constructors that may be executed in parallel	5.4.5
Synchronizer	Stop the workflow execution until the constructors have finished	5.4.6
Merge	Join directed edges into a single one	5.4.7
Counter	Determines the number of times that the execution flow goes through it	5.4.8
Condition	Evaluable expression that can be true or false	5.4.9
Operand	Object to be evaluated in a condition	5.4.10
Operator	Type of operation of a condition	5.4.11
LogicalOperator	Operator based on logical conditions	5.4.12
RelationalOperator	Operator based on relations	5.4.13
MathematicalOperator	Operator based on mathematical operations	5.4.14
ANDOperator	AND operator	5.4.15
OROperator	OR operator	5.4.16
NOTOperator	NOT operator	5.4.17
XOROperator	XOR operator	5.4.18
GreaterOperator	Greater operator	5.4.19
GreaterEqualOperator	Greater equal operator	5.4.20
LessOperator	Less operator	5.4.21
LessEqualOperator	Less equal operator	5.4.22
EqualOperator	Equal operator	5.4.23
NotEqualOperator	Not equal operator	5.4.24
SumOperator	Sum operator	5.4.25
SubtractionOperator	Subtraction operator	5.4.26
ProductOperator	Product operator	5.4.27
DivisionOperator	Division operator	5.4.28
ModuleOperator	Module operator	5.4.29

#### 5.4.1 Rationale

In some very specific data-intensive domains, the execution order of workflow elements is determined by the business logic. To represent these requirements, control-driven and controlled data-driven workflows need basic control structures, such as loops or conditional branching.

However, existing control-driven workflow languages are intended for defining business processes rather than data-intensive computing processes. These languages offer a vast number of elements since business process requirements involve many participants, gateways, events, asynchronous and synchronous communication, and different types of flows. In contrast, the control-driven requirements in data-intensive domains are typically more basic. The control flows

are limited to defining the start (*Begin*) and end (*End*) of the workflow, how to parallelise the execution flow (*Flow*, *Synchronizer* or *Merge*) or setting particular conditions to execute some programs (*Conditional*, *Condition*, *Operand* and *Operator*).

#### 5.4.2 Metaclass *Conditional*

A conditional defines a control structure that evaluates a boolean expression in order to determine the next constructor to execute. The expression can be evaluated as true or false.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

##### *Generalisation*

- ControlStructure (section 5.1.7)

##### *Specialisation*

N/A

##### *Attributes*

N/A

##### *Associations*

- evaluableExpression : Condition [1..\*]  
A boolean expression

##### *Constraints*

N/A

#### 5.4.3 Metaclass *Begin*

This metaclass defines the begin of the workflow definition. The constructors connected to this control structure will be the first constructors to be executed.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

##### *Generalisation*

- ControlStructure (section 5.1.7)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.4 Metaclass *End***

This metaclass defines the end of the workflow definition. The constructors connected to this control structure will be the last constructors to be executed.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ControlStructure (section 5.1.7)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.5 Metaclass *Fork***

This metaclass defines the separation of the execution flow into multiple branches that may be executed in parallel.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ControlStructure (section 5.1.7)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.6 Metaclass *Synchronizer***

This metaclass defines the join of the execution flow that was previously separated into multiple branches by a fork.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ControlStructure (section 5.1.7)

### ***Specialisation***

N/A

### ***Attributes***

- *maxSync* : int [1]  
The number of branches that have to be executed before continuing.
- *killer* : boolean [1]  
Stop the branched whose execution has not finished before continuing.

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.7 Metaclass *Merge***

This metaclass defines the join of the execution flow that was previously separated into multiple branches by a conditional control structure.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ControlStructure (section 5.1.7)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.8 Metaclass *Counter***

This metaclass defines a counter to keep track the number of times that the execution flow through it. A counter may be used to create explicit loops.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ControlStructure (section 5.1.7)

### ***Specialisation***

N/A

### ***Attributes***

- `init : int [1]`  
The initial value.
- `increment : int [1]`  
The value that will be added every time.
- `end : int [1]`  
The maximum value.

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.9 Metaclass *Condition***

A condition defines a boolean expression that produces a positive or negative value when evaluated.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Operand (section 5.4.10)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- `value:Operand [1..2]`  
The operands of the expression.
- `operation:Operator [0..1]`  
The type of expression.



### ***Constraints***

N/A

#### **5.4.10 Metaclass *Operand***

An operand defines the object of an evaluable expression.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

- Condition (section 5.4.9)
- Workflow::DataProvider (section 5.1.8)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.11 Metaclass *Operator***

An operator defines the type of an evaluable expression.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

- LogicalOperator (section 5.4.12)
- RelationalOperator (section 5.4.13)
- MathematicalOperator (section 5.4.14)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.12 Metaclass *LogicalOperator***

This metaclass define a type of operator based on logical conditions, such as AND, OR or XOR.

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Operator (section 5.4.11)

### ***Specialisation***

- ANDOperator (section 5.4.15)
- OROperator (section 5.4.16)
- NOTOperator (section 5.4.17)
- XOROperator (section 5.4.18)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

### 5.4.13 Metaclass *RelationalOperator*

This metaclass define a type of operator based on some kind of relation between the operands.

#### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

#### *Generalisation*

- Operator (section 5.4.11)

#### *Specialisation*

- GreaterOperator (section 5.4.19)
- GreaterEqualOperator (section 5.4.20)
- LessOperator (section 5.4.21)
- LessEqualOperator (section 5.4.22)
- EqualOperator (section 5.4.23)
- NotEqualOperator (section 5.4.24)

#### *Attributes*

N/A

#### *Associations*

N/A

#### *Constraints*

N/A

### 5.4.14 Metaclass *MathematicalOperator*

This metaclass define a type of operator based on mathematical operations.

#### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

#### *Generalisation*

- Operator (section 5.4.11)

### ***Specialisation***

- SumOperator (section 5.4.25)
- SubtractionOperator (section 5.4.26)
- ProductOperator (section 5.4.27)
- DivisionOperator (section 5.4.28)
- ModuleOperator (section 5.4.29)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.15 Metaclass *ANDOperator***

This metaclass defines an AND operator.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- LogicalOperator (section 5.4.12)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.16 Metaclass *OROperator***

This metaclass defines an OR operator.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- LogicalOperator (section 5.4.12)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.17 Metaclass *NOTOperator***

This metaclass defines a NOT operator.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- LogicalOperator (section 5.4.12)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.18 Metaclass *XOROperator***

This metaclass defines a XOR operator.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- LogicalOperator (section 5.4.12)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.19 Metaclass *GreaterOperator***

This metaclass defines an operator that evaluates if an operand value is greater than other.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- RelationalOperator (section 5.4.13)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.20 Metaclass *GreaterEqualOperator***

This metaclass defines an operator that evaluates if an operand value is equal or greater than other.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- RelationalOperator (section 5.4.13)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.21 Metaclass *LessOperator***

This metaclass defines an operator that evaluates if an operand value is less than other.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- RelationalOperator (section 5.4.13)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.22 Metaclass *LessEqualOperator***

This metaclass defines an operator that evaluates if an operand value is equal or less than other.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- RelationalOperator (section 5.4.13)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.23 Metaclass *EqualOperator***

This metaclass defines an operator that evaluates if an operand value is equal than other.



***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

***Generalisation***

- RelationalOperator (section 5.4.13)

***Specialisation***

N/A

***Attributes***

N/A

***Associations***

N/A

***Constraints***

N/A

**5.4.24 Metaclass *NotEqualOperator***

This metaclass defines an operator that evaluates if an operand value is different from other.

***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

***Generalisation***

- RelationalOperator (section 5.4.13)

***Specialisation***

N/A

***Attributes***

N/A

***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.25 Metaclass *SumOperator***

This metaclass defines an operator that adds the value of the operands.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- MathematicalOperator (section 5.4.14)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.4.26 Metaclass *SubtractionOperator***

This metaclass defines an operator that subtracts the value of the operands.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- MathematicalOperator (section 5.4.14)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.27 Metaclass *ProductOperator***

This metaclass defines an operator that multiplies the value of the operands.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- MathematicalOperator (section 5.4.14)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.28 Metaclass *DivisionOperator***

This metaclass defines an operator that divides the value of the operands.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- MathematicalOperator (section 5.4.14)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.4.29 Metaclass *ModuleOperator***

This metaclass defines an operator that finds the remainder after division of one operand value by another.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- MathematicalOperator (section 5.4.14)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.5 Package *DataTypes***

This package provides metaclasses to define the different data types that are supported by a scientific workflow (Figure 7). The list of supported data types is shown in Table 6.

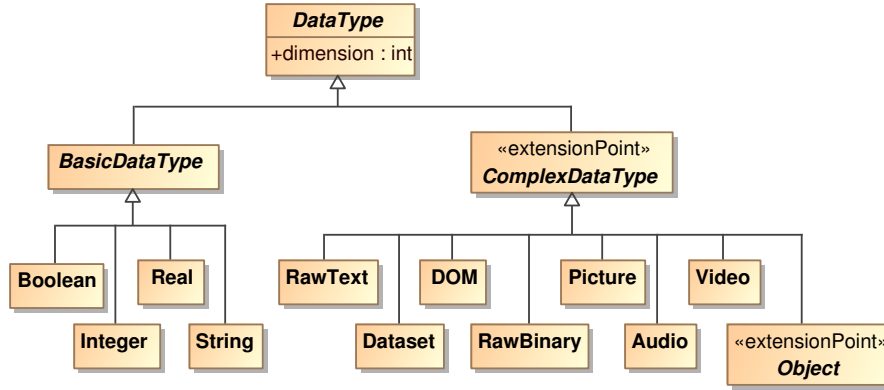


Figure 7: Data Types package

Table 6: Data Types package

Metaclass	Description	Section
DataType	Data type to be consumed or produced	5.5.2
BasicDataType	Primitive programming language data types	5.5.3
ComplexDataType	Non-primitive data types	5.5.4
Boolean	Boolean data type	5.5.5
Integer	Integer data type	5.5.6
Real	Real data type	5.5.7
String	String data type	5.5.8
RawText	Text-plan data type	5.5.9
Dataset	Table-like data type	5.5.10
DOM	Document object model data type	5.5.11
RawBinary	Binary-encoded data type	5.5.12
Picture	Image data type	5.5.13
Audio	Audio data type	5.5.14
Video	Video data type	5.5.15
Object	Programming language data type	5.5.16

### 5.5.1 Rationale

In order for a workflow element to be able to execute, it is necessary to specify the type of input and output data it can manage. All existing workflows support basic data types (*BasicDataType*) such as booleans (*Boolean*), strings (*String*), or numbers (*Integer* or *Real*). However, some complex data types (*ComplexDataType*) like pictures (*Picture*), videos (*Video*) or audio (*Audio*) are required in certain domains, such as video or audio processing, or image manipulation.

While most workflow tools support only basic data types for the execution of scientific workflows, some developers of workflow tools recognise the loss of domain semantics and allow for the definition of complex data types. To address this issue, we have incorporated some of the most commonly used complex data types in SWEL, and provided an extension point for users to define their own data types based on their specific needs.

### 5.5.2 Metaclass *DataType*

A data type defines the type of data that can be consumed or produced through a port. A data type that is equivalent to a primitive data type of most programming languages is denominated as basic data type. On the other hand, it is known as complex data type.

#### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

#### *Generalisation*

N/A

#### *Specialisation*

- BasicDataType (section 5.5.3)
- ComplexDataType (section 5.5.4)

#### *Attributes*

- dimension : int [1]  
Definition

#### *Associations*

N/A

#### *Constraints*

N/A

### 5.5.3 Metaclass *BasicDataType*

A basic data type is a primitive data type existing in the most of existing programming languages (e.g. Java, C, JavaScript), such as Integer, Boolean, String...

#### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

#### *Generalisation*

- DataType (section 5.5.2)

### ***Specialisation***

- Boolean (section 5.5.5)
- Integer (section 5.5.6)
- Real (section 5.5.7)
- String (section 5.5.8)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.5.4 Metaclass *ComplexDataType***

A complex data type is any data type that is not considered a basic data type. For instance, images, videos, audios...

### ***Metaproperties***

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

### ***Generalisation***

- *DataType* (section 5.5.2)

### ***Specialisation***

- *RawText* (section 5.5.9)
- *Dataset* (section 5.5.10)
- *DOM* (section 5.5.11)
- *RawBinary* (section 5.5.12)
- *Picture* (section 5.5.13)
- *Audio* (section 5.5.14)
- *Video* (section 5.5.15)
- *Object* (section 5.5.16)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.5.5 Metaclass *Boolean***

This metaclass defines a boolean data type.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- BasicDataType (section 5.5.3)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.5.6 Metaclass *Integer***

This metaclass defines an integer number data type.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- BasicDataType (section 5.5.3)



### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.5.7 Metaclass *Real***

This metaclass defines a float number data type.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- BasicDataType (section 5.5.3)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.5.8 Metaclass *String***

This metaclass defines a string data type.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- BasicDataType (section 5.5.3)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.5.9 Metaclass *RawText***

This metaclass defines a data type whose content is in text plain.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- ComplexDataType (section 5.5.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.5.10 Metaclass *Dataset***

This metaclass defines a data type whose content is structured as a dataset.

***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

***Generalisation***

- ComplexDataType (section 5.5.4)

***Specialisation***

N/A

***Attributes***

N/A

***Associations***

N/A

***Constraints***

N/A

**5.5.11 Metaclass *DOM***

This metaclass defines a data type whose content is bases in a document object model (e.g. XML or HTML).

***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

***Generalisation***

- ComplexDataType (section 5.5.4)

***Specialisation***

N/A

***Attributes***

N/A

***Associations***

N/A

### ***Constraints***

N/A

#### **5.5.12 Metaclass *RawBinary***

This metaclass defines a data type whose content is encoded in binary.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- ComplexDataType (section 5.5.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.5.13 Metaclass *Picture***

This metaclass defines a data type whose content is an image.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- ComplexDataType (section 5.5.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.5.14 Metaclass *Audio***

This metaclass defines a data type whose content is an audio.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- ComplexDataType (section 5.5.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.5.15 Metaclass *Video***

This metaclass defines a data type whose content is a video.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- ComplexDataType (section 5.5.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.5.16 Metaclass *Object***

This metaclass defines a data type whose content is an object of a particular programming language.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: Yes

### ***Generalisation***

- ComplexDataType (section 5.5.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.6 Package *ExceptionTypes***

This package provides metaclasses to define the different types of errors supported during the workflow execution (Figure 8), which are listed in Table 7.

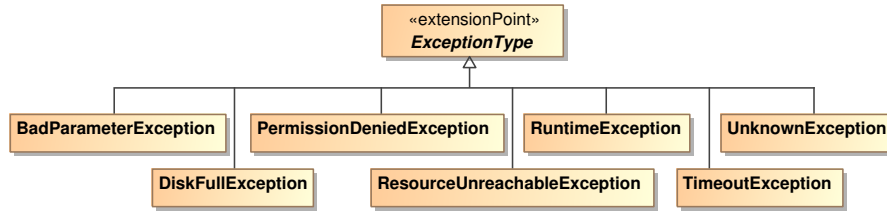


Figure 8: Exception Types package

Table 7: Exception Types package

Metaclass	Description	Section
ExceptionType	Error produced during work-flow execution	5.6.2
BadParameterException	Bad parameter error	5.6.3
DiskFullException	Disk full error	5.6.4
PermissionDeniedException	Permission denied error	5.6.5
ResourceUnreachableException	Resource unreachable error	5.6.6
RuntimeException	Runtime error	5.6.7
TimeoutException	Timeout error	5.6.8
UnknownException	Unknown error	5.6.9

### 5.6.1 Rationale

Since scientific workflows are intended for execution, error handling is an important issue in many workflow tools. Although most workflow tools do not support the specification of error handling and instead rely only on automatic management by the workflow engine, some particular workflow tools enable the definition of error handling to create more resilient and robust workflows.

In these scenarios, SWEL provides a set of elements to define an exception strategy. This strategy involves defining a set of actions to be taken when a particular type of exception is detected. Since the type of errors is directly related to the underlying computing platform, we provide an extension point to integrate and extend new specific exception types.

### 5.6.2 Metaclass *ExceptionType*

An exception type defines a type of error that is produced during the workflow execution. An error can be produced during the execution of a particular constructor or can be produced before constructor execution.

#### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

#### *Generalisation*

N/A

### ***Specialisation***

- `BadParameterException` (section 5.6.3)
- `DiskFullException` (section 5.6.4)
- `PermissionDeniedException` (section 5.6.5)
- `ResourceUnreachableException` (section 5.6.6)
- `RuntimeException` (section 5.6.7)
- `TimeoutException` (section 5.6.8)
- `UnknownException` (section 5.6.9)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.6.3 Metaclass *BadParameterException***

This metaclass defines that an error is caused by a bad configuration parameter.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- `ExceptionType` (section 5.6.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A



#### 5.6.4 Metaclass *DiskFullException*

This metaclass defines that an error is caused when there is not enough storage space.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

##### *Generalisation*

- ExceptionType (section 5.6.2)

##### *Specialisation*

N/A

##### *Attributes*

N/A

##### *Associations*

N/A

##### *Constraints*

N/A

#### 5.6.5 Metaclass *PermissionDeniedException*

This metaclass defines that an error is caused when access to a resource is forbidden.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

##### *Generalisation*

- ExceptionType (section 5.6.2)

##### *Specialisation*

N/A

##### *Attributes*

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.6.6 Metaclass *ResourceUnreachableException***

This metaclass defines that an error is caused when the access to a resource is unreachable.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- `ExceptionType` (section 5.6.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.6.7 Metaclass *RuntimeException***

This metaclass defines that an error is caused during the execution of a constructor.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- `ExceptionType` (section 5.6.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.6.8 Metaclass *TimeoutException***

This metaclass defines that an error is caused when a timeout has expired.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### ***Generalisation***

- `ExceptionType` (section 5.6.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.6.9 Metaclass *UnknownException***

This metaclass defines that an error is caused for unknown reasons.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: Yes
- *isExtensible*: No

### Generalisation

- ExceptionType (section 5.6.2)

### Specialisation

N/A

### Attributes

N/A

### Associations

N/A

### Constraints

N/A

## 5.7 Package *ComputationalSpecification*

This package provides metaclasses to define the different types of computational resources that have to be satisfied to execute a computational task (Figure 9). This information is intended to optimize the workflow execution, therefore is not required to be included in a workflow definition. All computational resources supported by SWEL are listed in Table 8.

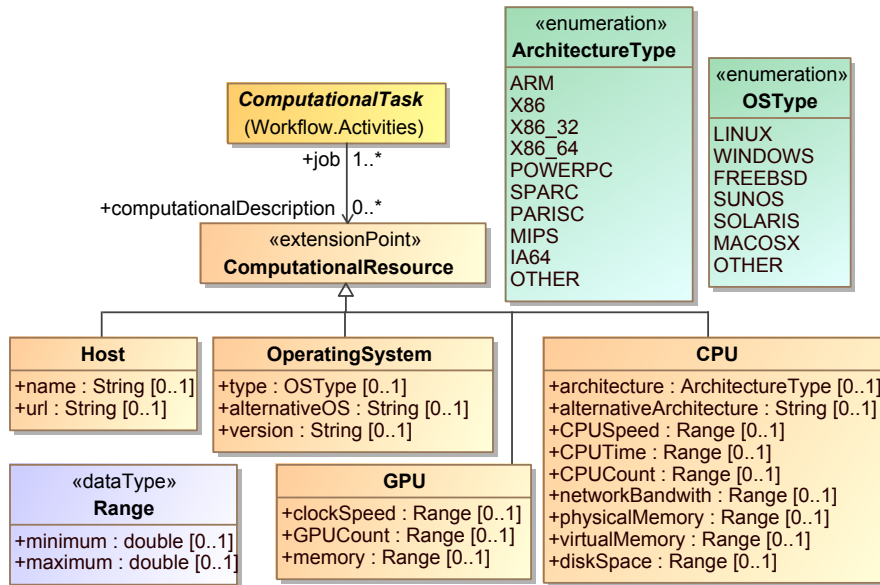


Figure 9: Computational Specification package

Table 8: Computational Specification package

Metaclass	Description	Section
ComputationalResource	Type of computational resource	5.7.2
Host	Computer accessible through network	5.7.3
OperatingSystem	Type of operating system	5.7.4
CPU	Type of CPU	5.7.5
GPU	Type of GPU	5.7.5
OSType	Enumeration of operating systems	5.7.7
ArchitectureType	Enumeration of CPU architectures	5.7.8

### 5.7.1 Rationale

In certain DI domains where high-performance computing is required, it is necessary to include this type of information in the definition of DIW. Workflow engines use underlying computing platforms to optimise workflow execution. It is worth noting that this information is not typically defined by humans but rather generated by workflow engines and associated with the workflow definition to ensure optimisation and reproducibility.

Furthermore, it has been observed that in some data-intensive domains, the use of available computing resources is a critical concern. For this reason, it is necessary to define the minimum or recommended computational resources. These resources generally relate to the use of particular CPUs (*CPU* and *ArchitectureType*), GPUs (*GPU*) or operating systems (*OperatingSystem* and *OSType*).

It should also be noted that the computational specification provided by SWEL is based on the Job Submission Description Language (JSDL). JSDL is an extensible XML specification from the Global Grid Forum designed for the description of simple tasks to non-interactive computer execution systems.

### 5.7.2 Metaclass *ComputationalResource*

This metaclass defines a type of computational resource.

#### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: Yes

#### *Generalisation*

N/A

#### *Specialisation*

- Host (section 5.7.3)
- OperatingSystem (section 5.7.4)
- CPU (section 5.7.5)
- GPU (section 5.7.6)

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.7.3 Metaclass *Host***

A host defines the information required to locate a specific machine in the network.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ComputationalResource (section 5.7.2)

### ***Specialisation***

N/A

### ***Attributes***

- name : String [0..1]  
The name of the machine in the network.
- url : String [0..1]  
The URL to access the machine.

### ***Associations***

N/A

### ***Constraints***

- “name” or “url” must be defined.

## **5.7.4 Metaclass *OperatingSystem***

An operating system defines the basic features of an operating system.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ComputationalResource (section 5.7.2)

### ***Specialisation***

N/A

### ***Attributes***

- type : OType [0..1]  
The type of operating system.
- alternativeOS : String [0..1]  
An alternative, compatible operating system.
- version : String [0..1]  
The version of the operating system.

### ***Associations***

N/A

### ***Constraints***

N/A

## **5.7.5 Metaclass *CPU***

This metaclass defines the configuration of a CPU.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ComputationalResource (section 5.7.2)

### ***Specialisation***

N/A

### ***Attributes***

- architecture : ArchitectureType [0..1]  
The type of architecture.
- alternativeArchitecture : String [0..1]  
An alternative, compatible architecture.
- CPUSpeed : Range [0..1]  
The clock speed of the CPU.
- CPUTime : Range [0..1]  
The number of CPU time seconds that is allowed to consume.
- CPUCount : Range [0..1]  
The number of CPUs.
- networkBandwidth : Range [0..1]  
The bandwidth available in the network.
- physicalMemory : Range [0..1]  
The amount of physical memory.
- virtualMemory : Range [0..1]  
The amount of virtual memory.
- diskSpace : Range [0..1]  
The amount of disk space.

### ***Associations***

N/A

### ***Constraints***

N/A

### **5.7.6 Metaclass *GPU***

This metaclass defines the configuration of a GPU.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- ComputationalResource (section 5.7.2)

### ***Specialisation***

N/A



### ***Attributes***

- clockSpeed : Range [0..1]  
The clock speed of the GPU.
- GPUCount : Range [0..1]  
The number of GPU cores.
- memory : Range [0..1]  
The amount of physical memory.

### ***Associations***

N/A

### ***Constraints***

N/A

#### **5.7.7 Enumeration *OSType***

OSType is an enumeration type that specifies the literals for defining the type of compatible operating system

### ***Values***

- LINUX  
Indicates that the operating system is based on Linux.
- WINDOWS  
Indicates that the operating system is based on Windows.
- FREEBSD  
Indicates that the operating system is based on FreeBSD.
- SUNOS  
Indicates that the operating system is based on SunOS.
- SOLARIS  
Indicates that the operating system is based on Solaris.
- MACOSX  
Indicates that the operating system is based on MacOSX.
- OTHER  
Indicates that the operating system is unknown.

#### **5.7.8 Enumeration *ArchitectureType***

OSType is an enumeration type that specifies the literals for defining the type of architecture

### *Values*

- ARM  
Indicates that the architecture is based on ARM.
- X86  
Indicates that the architecture is based on X86.
- X86\_32  
Indicates that the architecture is based on X86\_32.
- X86\_64  
Indicates that the architecture is based on X86\_64.
- POWERPC  
Indicates that the architecture is based on POWERPC.
- SPARC  
Indicates that the architecture is based on SPARC.
- PARISC  
Indicates that the architecture is based on PARISC.
- MIPS  
Indicates that the architecture is based on MIPS.
- IA64  
Indicates that the architecture is based on IA64.
- OTHER  
Indicates that the architecture is unknown.

## 6 Specification layer

### 6.1 Package *WFSpecification*

This package provides metaclasses to define meta-information related to the scientific workflow (Figure 10). This information is related to the human resources (Table 9) involved in the design and execution of the workflow.

Table 9: WFSpecification package

Metaclass	Description	Section
WFSpecification	Metadata related to a workflow	6.1.2
Project	Project that led the design of a workflow	6.1.3
Stakeholder	Participant in a project	6.1.4
Person	Human entity	6.1.5
Organization	Entity composed by multiple people	6.1.6

#### 6.1.1 Rationale

In collaboration with data scientists, we recognised the significance of incorporating information about the project or scientific experiment that motivated the creation of a DIW. This information (*WFSpecification*) provides a broader

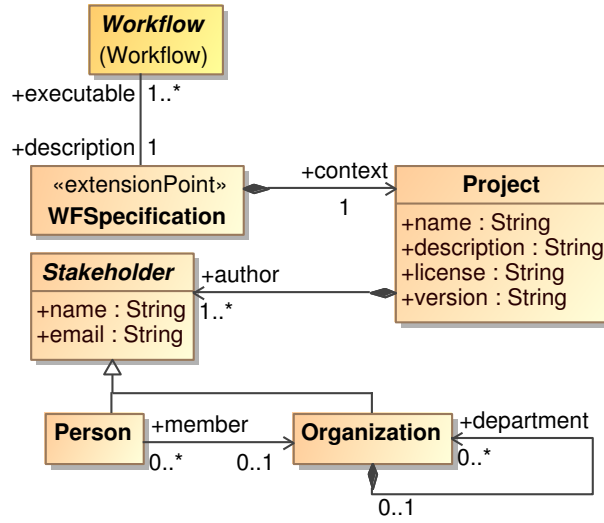


Figure 10: WFSpecification package

context for the workflow’s design and objectives, making it easier to comprehend the outcomes after execution and facilitate knowledge transfer between specialists.

This need arose while working with DIWs from repositories like myExperiment.org, which include this information to provide context for the workflows (*Project* and *Stakeholder*). However, it is important to note that this information is not directly linked to the DIW and is simply metadata that helps preserve the authors’ description of its behaviour once the DIW is downloaded. In addition to external repositories, we also considered the requirements of DIWs developed by the domain experts. Therefore, we explored potential languages for defining scientific experiments in DI to adapt a representation compatible with other proposals in SWEL.

### 6.1.2 Metaclass *WFSpecification*

This metaclass defines the metadata related to the workflow. This information is intended to be read by humans, not to be processed by computers.

#### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

#### *Generalisation*

N/A

#### *Specialisation*

N/A

### ***Attributes***

N/A

### ***Associations***

- context : Project [1]  
The information about the context that led the design of the workflow.

### ***Constraints***

N/A

### **6.1.3 Metaclass *Project***

A project defines the information about the context in which the workflow has been designed. This information is intended to be only consumed by humans.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

N/A

### ***Specialisation***

N/A

### ***Attributes***

- name : String [1]  
Definition
- description : String [1]  
Definition
- license : String [1]  
Definition
- version : String [1]  
Definition

### ***Associations***

- author : Stakeholder [1..\*]  
Definition

### ***Constraints***

N/A

#### 6.1.4 Metaclass *Stakeholder*

A stakeholder defines the main information about an entity that is participating in the project.

##### *Metaproperties*

- *isAbstract*: Yes
- *isFinal*: No
- *isExtensible*: No

##### *Generalisation*

N/A

##### *Specialisation*

- Person (section 6.1.5)
- Organization (section 6.1.6)

##### *Attributes*

- name : String [1]  
The name of the entity.
- email : String [1]  
The email address of the entity.

##### *Associations*

N/A

##### *Constraints*

N/A

#### 6.1.5 Metaclass *Person*

A person specifies a human entity.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

##### *Generalisation*

- Stakeholder (section 6.1.4)

##### *Specialisation*

N/A

### ***Attributes***

N/A

### ***Associations***

- `memberOf` : Organization [0..1]  
Definition

### ***Constraints***

N/A

## **6.1.6 Metaclass *Organization***

An organization specifies an entity comprising multiple people, such as an institution or an association.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Stakeholder (section 6.1.4)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

- `department` : Organization [0..\*]  
List of departments, structured as organizations, that compose the organization.

### ***Constraints***

N/A

## **6.2 Package *ExperimentSpecification***

This package provides metaclasses to define meta-information about the scientific experiment (Figure 11). This information is not intended to be used during workflow execution, but it is used as human-readable documentation about the problem to solve by the scientific workflow.

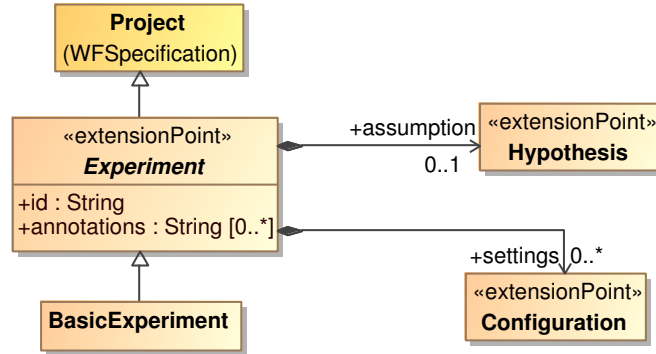


Figure 11: Experiment Specification package

Table 10: Experiment Specification package

Metaclass	Description	Section
Experiment	Type of project to accept or reject an hypothesis	6.2.2
BasicExperiment	Experiment with the basic parameterization	6.2.3
Hypothesis	Problem to be test it	6.2.4
Configuration	Parameters and initial settings	6.2.5

### 6.2.1 Rationale

We have observed that many DIWs are implemented in the context of conducting a scientific experiment (*Experiment*) to either validate or reject a hypothesis (*Hypothesis*). This information is particularly important for gaining a deeper understanding of the workflow execution results, the specific experimental configuration (*Configuration*), and is necessary for many scientific DI domains.

Given that SWEL is not intended for defining scientific experiments, we offer a range of extension points that enable SWEL users to expand language and adapt it to their specific needs. We believe that this information should be included in the workflow definition to provide easily understandable information and enable the integration of SWEL with scientific tools.

### 6.2.2 Metaclass *Experiment*

An experiment defines a type of project to support, refute, or validate a hypothesis.

#### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: Yes

#### *Generalisation*

- Project (section 6.1.3)

### ***Specialisation***

- BasicExperiment (section 6.2.3)

### ***Attributes***

- id : String [1]  
A value that identifies the experiment.
- annotations : String [0..\*]  
List of notes intended to be read by humans.

### ***Associations***

- assumption : Hypothesis [0..1]  
The assumption to be accepted or refused by an experiment.
- settings : Configuration [0..\*]  
The set of parameters required to perform an experiment.

### ***Constraints***

N/A

## **6.2.3 Metaclass *BasicExperiment***

A basic experiment specifies a type of experiment not requiring special parametrisation or information.

### ***Metaproperties***

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: No

### ***Generalisation***

- Experiment (section 6.2.2)

### ***Specialisation***

N/A

### ***Attributes***

N/A

### ***Associations***

N/A

### ***Constraints***

N/A



#### 6.2.4 Metaclass *Hypothesis*

A hypothesis defines a problem that has to be tested by a particular experiment. The hypothesis can be accepted or rejected.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: Yes

##### *Generalisation*

N/A

##### *Specialisation*

N/A

##### *Attributes*

N/A

##### *Associations*

N/A

##### *Constraints*

N/A

#### 6.2.5 Metaclass *Configuration*

Configuration defines the set of parameters and initial settings of an experiment.

##### *Metaproperties*

- *isAbstract*: No
- *isFinal*: No
- *isExtensible*: Yes

##### *Generalisation*

N/A

##### *Specialisation*

N/A

##### *Attributes*

N/A

***Associations***

N/A

***Constraints***

N/A

# Appendices

## APPENDICES

### A Exemplary graphical and textual concrete syntax

Although SWEL is platform and notation independent, its elements can be mapped to a graphical or textual concrete syntax to make the definition of DI workflows usable by application domain experts and exploitable by workflow tools. In this section, we present two illustrative examples of such mapping to concrete notations, one graphical and one textual. For the former, Table 11 shows the subset of SWEL that will be represented graphically. The column SWEL *type* indicates the name of the abstract metaclass that groups related concrete metaclasses. The column SWEL *element* shows the name of the language metaclass to be mapped to a particular *Graphical element*.

Table 11: Partial examples of concrete syntax.


















SWEL type	SWEL element	Graphical element
<i>DirectedEdge</i>	ControlLine	
	DataLine	
	ExceptionLine	
<i>ControlStructure</i>	Begin	
	End	
	Conditional	
	Fork	
	Synchronizer	
	Merge	
	Counter	
<i>Activity</i>	ComputationalTask	
	DisplayTask	
<i>DataProvider</i>	Record	
	File	
	Stream	
	Database	
<i>Experiment</i>	BasicExperiment	

Figure 12 illustrates the use of this graphical notation for a DIW definition taken from a public workflow repository <sup>1</sup>. This is an excerpt of a data-driven workflow that counts the number of publications and citations per year for one author from a particular biomedical information service. First, it searches for publications (computational tasks *searchPublications.input* and *searchPublications*) to retrieve bibliographic records for all the papers published by a single author (record *author\_name*). Next, it extracts the citations to get the year of all the papers being cited (computational task *extract\_citation\_pubYear*). The

<sup>1</sup>myExperiment. <https://www.myexperiment.org>

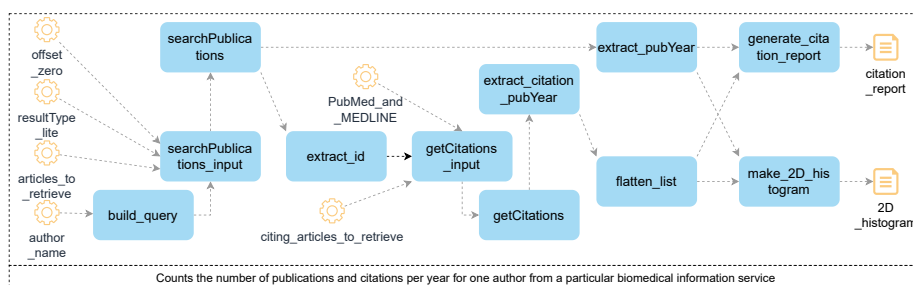


Figure 12: SWEL representation of a data-driven workflow.

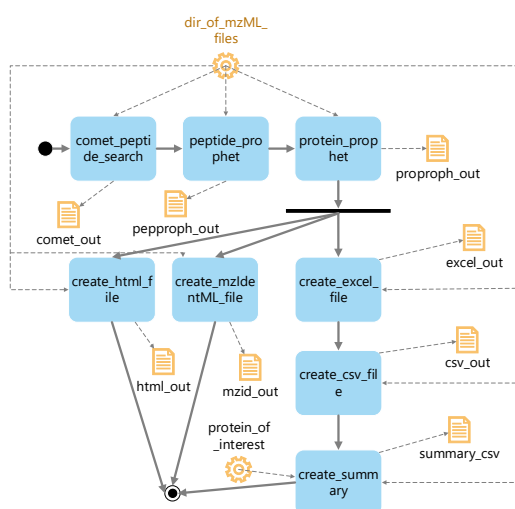


Figure 13: SWEL representation of a control-driven workflow.

publication year is extracted (computational task *extract\_pubYear*) from the raw web service results. Both citations and publication years are sent to both a display task to produce 2D histograms as output and a task generating a report that shows the number of citations per year. Note that control structures are not used as it is a data-driven workflow. Thus, data dependencies, along with the corresponding input and output ports, will be considered by the workflow engine to determine the execution order and the underlying data composition patterns.

In the case of Figure 13, the same notation is used for the representation of the control-driven workflow, which aims to perform a search from a particular service and a subsequent validation of the results that are finally exported to files of different formats. In contrast to the previous example, this illustrates a DI application in a domain with process-centric requirements. Therefore, a list of control structures, such as fork or synchronisation, which defines the execution order according to the business goals, is used instead of data dependencies and composition.

Listing 1 shows a snippet of the workflow represented graphically in Figure 12, but using a JSON-based textual notation for SWEL instead. With this syntax, a first level defines the project meta-information of the *Specification* package. Then, a second level will be given by the definition of the DIWs that

make up this project, including the nodes (type, attributes, configuration, notes, etc.) and the control or data links between them. For the example case, this JSON code snippet defines a data-driven DIW (line 1) with two nodes (line 2): a record (line 3) that is assigned a name (line 4) and its respective value (line 5), and a web service (lines 6–8) with information about the particular required operation (line 9). Both are linked by a data link (lines 11–13).

Listing 1: Code snippet of a JSON-based notation for SWEL

```

1 ... {"type": "dataDrivenWorkflow",
2     "nodes": [{
3         "type": "record",
4         "name": "citing_articles_to_retrieve",
5         "value": "100"
6     }, {
7         "type": "soapClient",
8         "name": "searchPublications",
9         "url": "http://www.ebi.ac.uk/(/..)/soap?wsdl",
10        "operation": "searchPublications" }, ...],
11    "links": [{
12        "type": "data",
13        "source": "extract_id",
14        "target": "getCitations_input" }, ...] }
```

## B Illustrative matching with existing concrete syntax

SWEL is not coupled to any particular concrete syntax to match the different language elements to a particular type of representation, neither textual nor graphic. Therefore, since SWEL is independent of any platform and tool, multiple concrete syntaxes can be used to define SWEL workflows.

Two different tools have been used to illustrate that SWEL can be matched to multiple concrete syntaxes. Table 12 gathers the corresponding relationship between some of SWEL metaclasses and the visual representation of the main elements of Taverna. Taverna concrete syntax uses a row-based box to represent activities, whose background colour determines the type of activity. The first row is divided according to the number of input data, the second row is the name of the activity and, finally, the last row is composed of the different outputs. On the other hand, the data providers are represented by a single box that contains the name of the input or output. Finally, the data lines are matched to simple arrows.

Table 13 shows how the main LONI Pipeline elements can be matched to some of the SWEL metaclasses. In this case, an activity is represented by a big circle, whose upper little circles are the activity inputs, and the lower little triangles are the activity outputs. The conditional element is matched to a rounded rectangle, which contains a rhombus icon with a rounded rectangle to show the name. The data providers are represented by a circle with a lower little triangle if the data provider is an input of data, or by a triangle with a upper little circle if the data provider is an output of data. Finally, the data line is a simple line.

## C Workflow examples with SWEL

In this section, some scientific workflows are defined using SWEL. To represent such workflows, both concrete syntaxes introduced above are used.

Table 12: Illustrative matching with partial Taverna concrete syntax

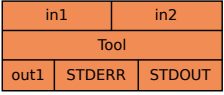
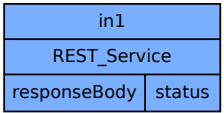
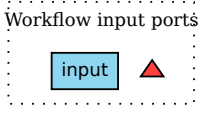
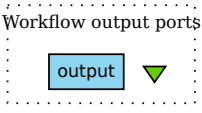








Metaclass	Representation
Activities::CliProcess	
Activities::RestClient	
DataProviders::Record (Input)	
DataProviders::Record (Output)	
ExecutionGraph::DataLine	

Table 13: Illustrative matching with partial LONI Pipeline concrete syntax

Metaclass	Representation
Activities::CliProces	
ControlStructures::Conditional	
DataProviders::File (Input)	
DataProviders::Record (Input)	
DataProviders::File (Output)	
DataProviders::Record (Output)	
ExecutionGraph::DataLine	

## C.1 Discover diseases

This workflow is intended to find potential diseases from a set of keywords introduced by the user. After retrieving a set of medical documents according to specific parameters, they are processed to extract a list of proteins associated with different diseases, which are used to discover new diseases related to the given keywords. Figure 14 shows the representation of this workflow using the Taverna concrete syntax presented above, and Figure 15 illustrates the same workflow but using the LONI Pipeline concrete syntax described previously.

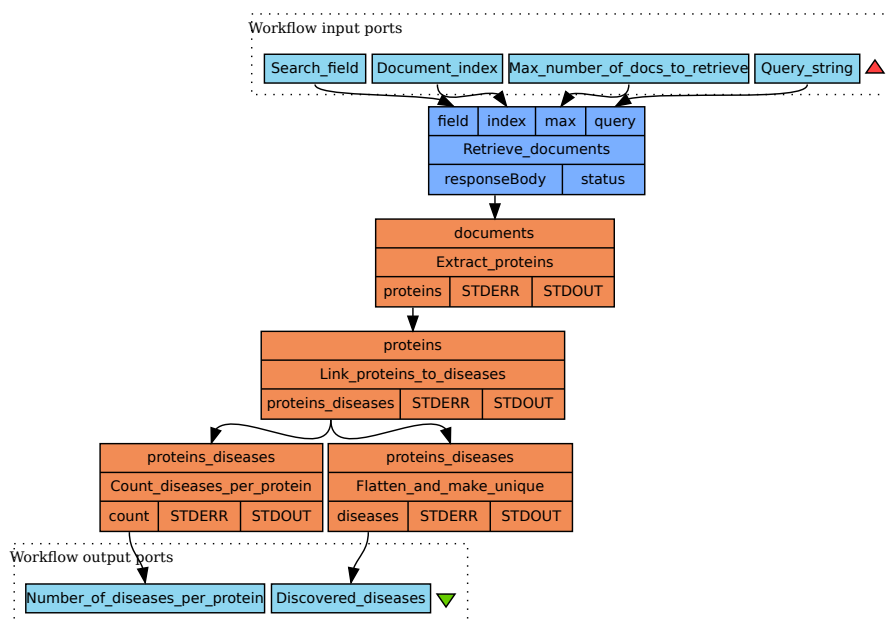


Figure 14: Workflow using Taverna concrete syntax

## C.2 Outlier detection in medical claims

This workflow is focused on detecting outliers, i.e. the identification of claims with an unusually high cost for a specific disease. After acquiring data, they are preprocessed and then those claim records, whose cost deviates from the average value of the group they belong to, are identified and returned. Figure 16 depicts this workflow using the Taverna concrete syntax, and in Figure 17 the workflow is represented using the LONI Pipeline concrete syntax.

## D Collaborative workflow definitions

In many cases, the definition of DIW is a collective effort, where professionals from different disciplines and organisations work to reach a common goal, typically using heterogeneous workflow tools. In such situations, a mechanism to propagate and update changes consistently is required to keep synchronisation between all participants and tools involved in the workflow definition. Unfortunately, such synchronisation is usually achieved manually, which is an error-prone process and does not allow instant or automatic updates. In our

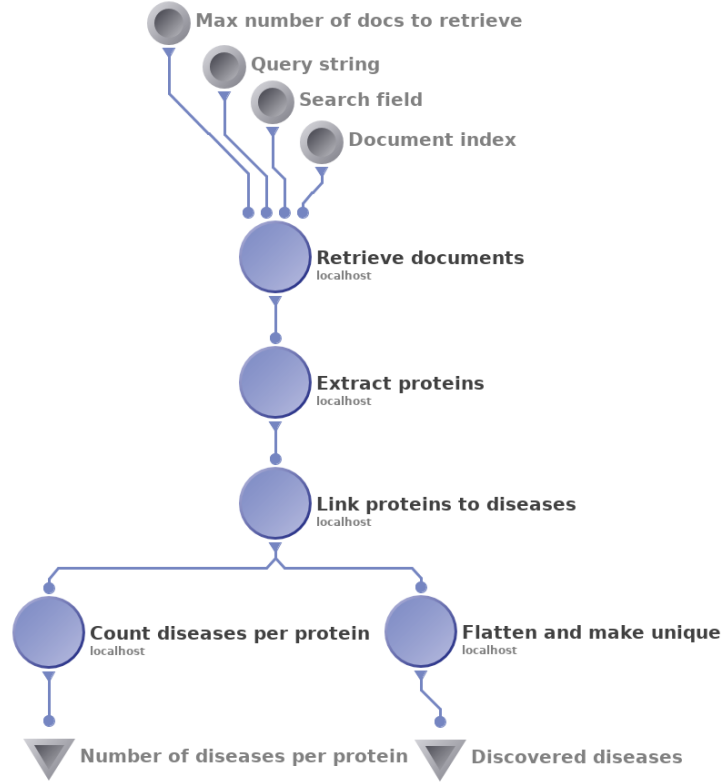


Figure 15: Workflow using LONI Pipeline concrete syntax

proposal, bidirectional model transformations (BX) can be defined to propagate the changes on the source models to the target models, and vice-versa, automatically maintaining the synchronisation between the models. Instead of developing separate transformations in both directions, BX permit reducing development time and avoiding the task of designing consistent unidirectional transformations.

More interestingly, BX allows keeping in sync the same workflow definition declared in two different workflow languages, such as SWEL and CWL. A simple workflow is used to demonstrate how SWEL enables collaborative workflow definition. Namely, the workflow depicted in Figure 18 computes how many times a word appears in a particular document by executing two tasks: process *get-document* accesses the document via a URL (record *url*) and stores its content in a text file. Then, process *count-document-words* reads the file and extracts the list of words together with the number of their occurrences. The resulting list is stored in a file text, *count*.

For this case study we have defined a QVT bidirectional transformations between SWEL and CWL. Given that CWL does not have a formalised metamodel, a CWL metamodel has been manually defined according to the requirements of this particular application case. Listing 2 illustrates the *MapWorkflow-Commandlinetool2CliProcess* QVT relation, which creates the correspondences between console commands in CWL (lines 4–6) and SWEL (lines 7–9).

Note that in the relation both CWL (CommandLineTool) and SWEL (CliProcess) have been added as “enforce domain” to allow QVT to perform the transformation in both directions. In contrast, making use of the formulation “check-



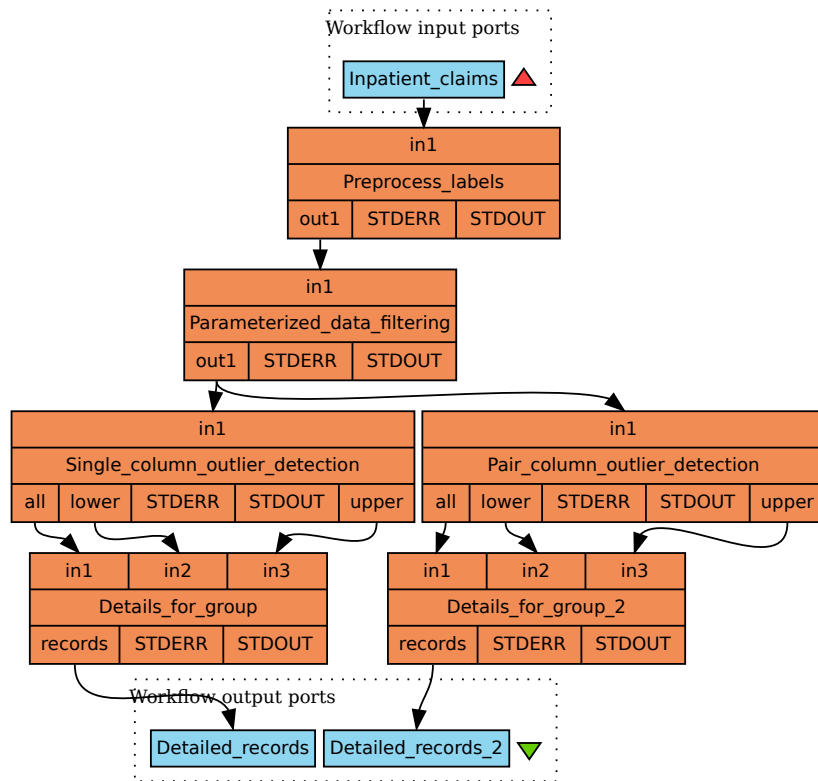


Figure 16: Workflow using Taverna concrete syntax

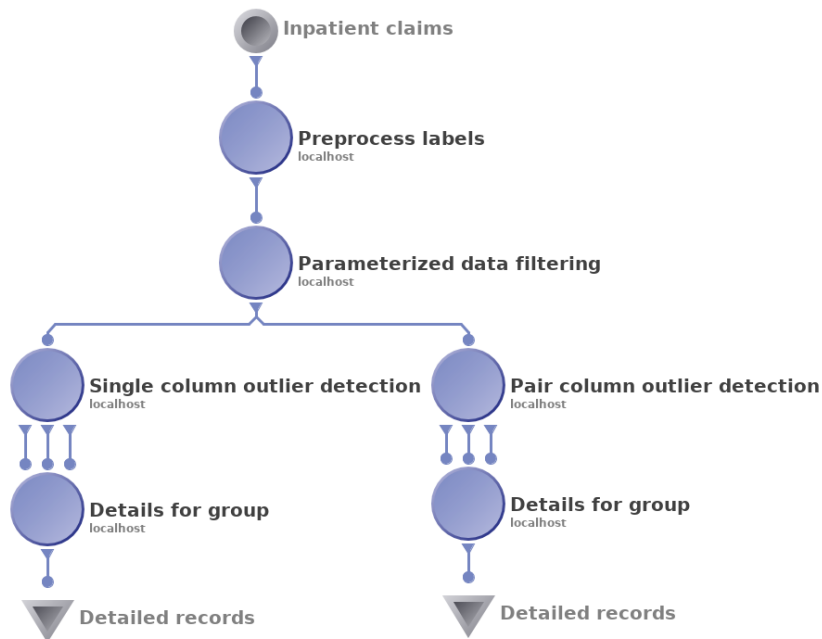


Figure 17: Workflow using LONI Pipeline concrete syntax

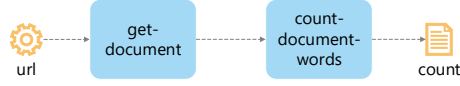


Figure 18: SWEL simple workflow representation.

only domain” at one end denotes that this point is read-only and, consequently, the relationship is unidirectional.

Listing 2: QVT bidirectional relation: *MapWorkflowCommandLineTool2CliProcess*

```

1 relation MapWorkflowCommandLineTool2CliProcess {
2   varName : String;
3   varCommand : String;
4   enforce domain cwl clt : commonwl::CommandLineTool
5   {
6     name = varName,
7     baseCommand = varCommand };
7   enforce domain swel cp : SWEL::Workflow::Activities::CliProcess
8   {
9     name = varName,
10    executable = varCommand }; }

```

It should be noted that reaching full consistency between DIW is directly dependent on the technology used for bidirectional transformations. This is a field with active research, so many of the current tools are prototypes or still unstable.