Bus Inf Syst Eng 66(2):137–160 (2024)
https://doi.org/10.1007/s12599-023-00826-7

RESEARCH PAPER

SWEL: A Domain-Specific Language for Modeling Data-Intensive Workflows

Rubén Salado-Cid · Antonio Vallecillo · Kamram Munir · José Raúl Romero

Received: 4 April 2022 / Accepted: 20 June 2023 / Published online: 6 August 2023

Abstract Data-intensive applications aim at discovering valuable knowledge from large amounts of data coming from real-world sources. Typically, workflow languages are used to specify these applications, and their associated engines enable the execution of the specifications. However, as these applications become commonplace, new challenges arise. Existing workflow languages are normally platform-specific, which severely hinders their interoperability with other languages and execution engines. This also limits their reusability outside the platforms for which they were originally defined. Following the Design Science Research methodology, the paper presents SWEL (Scientific Workflow Execution Language). SWEL is a domain-specific modeling language for the specification of data-intensive workflows that follow the model-driven engineering principles, covering the high-level definition of tasks, information sources, platform requirements, and mappings to the target technologies. SWEL is platform-independent, enables collaboration among data scientists across multiple domains and facilitates interoperability. The evaluation results show that SWEL is suitable enough to represent the concepts and mechanisms of commonly used data-intensive workflows. Moreover, SWEL facilitates the development of related technologies such as editors, tools for exchanging knowledge assets between workflow management systems, and tools for collaborative workflow development.

Keywords Model-driven engineering · Domain-specific modeling · Conceptual modeling · Data-intensive applications · Data-driven workflows · Data science

Accepted after 2 revisions by Pnina Soffer.

R. Salado-Cid · J. R. Romero (✉)
Department of Computer Science and Numerical Analysis, University of Córdoba, Córdoba, Spain
e-mail: jrromero@uco.es

A. Vallecillo
ITIS Software, Universidad de Málaga, Málaga, Spain

K. Munir
FET - Computer Science and Creative Technologies, University of the West of England, Bristol, UK

1 Introduction

The amount of data collected by companies and organizations is growing exponentially (Szalay and Gray 2006; Buhl et al. 2013), as they want to make the most of it by extracting useful new knowledge. In this context, the so-called data-intensive (DI) applications (Chen and Zhang 2014) aim at discovering valuable knowledge from huge amounts of data coming from real-world sources. These applications are becoming common in many domains including, e.g., e-commerce, financial markets, manufacturing, marketing, education, or social sciences (Tera Allas et al. 2018). The scientific sector is particularly interested in DI applications (van der Aalst and Damiani 2015; Demchenko et al. 2013) because many research areas have become highly data-driven, such as bioinformatics, astronomy, or healthcare (Chen and Zhang 2014).

Regardless of the field of application, the data management and knowledge discovery processes of any DI application are normally formulated as a pipeline. Here, the pipeline contains sequences 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 visualization. The representation of such pipelines as data-intensive workflows (DIW), a.k.a. scientific workflows (Coleman et al. 2022), enables high-level definition of these processes. It also improves the understanding of the DI tasks by those professionals who are not skilled in data science but are experts in their respective domains (Salado-Cid et al. 2018). In essence, DIW bridges between data scientists, domain specialists, and the target computing infrastructure (Sethi and Gil 2017).

General-purpose programming languages like Python or statistical frameworks like R have been traditional solutions to code pipelines in data science, but they require high computing skills. In turn, existing DIW languages allow defining the specific sequence of tasks to be completed (*what*), but not *where* these tasks should be executed, nor *how* the associated resources should be arranged. Examples of such languages include DAX for Pegasus (Deelman 2015), SCUFL for Taverna (Oinn et al. 2004), MoML for Kepler (Altintas et al. 2004), or AGWL for ASKALON (Fahringer et al. 2004). Moreover, they are often tied to specific DIW management systems (WfMS) (Yu and Buyya 2006), something that hampers their interoperability with other workflow languages and execution engines. This limits the reusability of their artifacts outside the platforms which they were defined for, thereby hindering collaboration among data scientists (Coleman et al. 2022).

To address these limitations, a few platform-independent language proposals in this field attempt to provide a complete set of components and elements, so that any tool can make use of their constructs to define workflows. A first proposal was the Abstract Grid Workflow Language (AGWL) (Fahringer et al. 2004), which is an XML-based notation for describing grid workflow applications independent of implementation details. More recently, the Common Workflow Language (CWL) (Amstutz et al. 2020) enables a complete, multi-vendor specification of DIW, which fully describes the data and execution pipelines, and is supported by several JSON-based utilities and tools. Nevertheless, both initiatives are still technology-dependent and rely on the definition of a concrete, parseable syntax. This restricts their applicability to only those tools that meet their technological and language requirements.

To tackle these issues, this paper presents SWEL (Scientific Workflow Execution Language), a domain-specific modeling language (DSML) (Bucchiarone et al. 2021) for the abstract specification of data, execution, and experimentation pipelines. SWEL is independent of any tool or platform, it enables collaboration among data scientists by reusing the knowledge across domains and facilitates interoperability between tools. SWEL covers the whole DIW specification, from the high-level definition of the problem in terms of the DI tasks to be performed, the sources of information, the platform requirements, and the mappings to the target execution technologies.

The methodological approach used in this work is *Design Science Research* (DSR) (Johannesson and Perjons 2014). SWEL is the main artifact developed resulted from a cycle of activities leading to solving the real-world problem posed above. The problem addressed in this paper originated in the context of the software company where one of the authors works. The company develops scientific workflows for user data analysis using different languages. There was a real need for a suitable and generic platform-independent workflow language that would allow the reuse of previous developments made in other languages.

To achieve its purpose, SWEL is built according to the precepts of model-driven engineering (MDE) (Brambilla et al. 2017). Its concepts and components are formally modeled by means of *metamodels* that describe the language elements, the relations among them, and their constraints and governing rules. These metamodels are not tied to any concrete syntax or technological platform, but bridges to them can be easily defined and implemented using model transformations and current MDE tools. In this paper, we illustrate the use of model transformations to achieve interoperability between DIW tools, and show the applicability and suitability of the language to enable collaboration across multiple DI domains. Moreover, SWEL can be extended to capture certain domain-specific concepts when needed.

The rest of the paper is organized in accordance with the DSR methodology. Section 2 defines the state of the art, including some background on DIW and DSML, and introduces key related work on DIW languages. Section 3 explains the DSR methodology and outlines the contribution requirements. The design and overall architecture of SWEL, as well as the precise metamodels of the main resulting artifact (SWEL) are presented in Sect. 4. Section 5 demonstrates how SWEL can be used to develop practicable toolkits: a JSON concrete syntax validator and an editing tool for creating workflows using a graphical notation. As part of the DSR cycle, a case study has been developed to validate SWEL and illustrate its applicability. This is discussed in Sect. 6. Section 7 evaluates SWEL as a pivot language for achieving interoperability of existing workflow languages, including a discussion on the threats to the validity. Finally, Sect. 8 draws important conclusions and outlines future extensions to this work.

2 State of the Art

This section reviews the fundamentals of DSMLs and DIWs, the two main areas of knowledge required for the understanding and development of SWEL. Then, it dives into the current forms of DIW-specific languages.

2.1 Domain-Specific Modeling Languages

Domain-specific languages (DSL) allow practitioners to represent their reality with language constructs that they perceive as closer to their domain. The goal is to improve productivity and communication among domain specialists, who can focus on what a task should do, instead of how it should be performed (Fowler 2010). In general, DSL is a term encompassing both domain-specific "programming" languages, i.e., those that are defined at the implementation level according to a given grammar or structure, and DSMLs. For example, WfMS can serialise workflows to their respective workflow languages (Bucchiarone et al. 2021), i.e., some DSL formalizing the set of elements required to specify tasks, links, constraints and resources of the workflow-based pipeline for that particular tool. In contrast, a DSML is a particular type of DSL whose definition is given in terms of models which are closer to the problem domain than to the implementation domain. This makes the language independent of specific platforms or technologies and, consequently, abstracts away implementation detail, thus avoiding adding accidental complexity (Brambilla et al. 2017).

A DSML consists of three main components (Bucchiarone et al. 2021): (1) the abstract syntax is a model (called *metamodel*) defining the language concepts and their relationships, as well as the governing rules that constrain the domain; (2) the concrete syntax (or notation) enables the realisation of the language elements in terms of textual or graphical symbols; and (3) the semantics refers to the meaning of the language elements, consistently and precisely expressed. Examples of DSMLs include iContractML (Hamdaqa et al. 2020) for modeling and deploying smart contracts, BoSDL (Schlauderer and Overhage 2018) for describing business-oriented software services, Model4CEP (Boubeta-Puig et al. 2015) for the definition of complex event processing, or DSML4CSR (Campos and Grangel 2018) for corporate social responsibility.

A recurrent issue with DSLs refers to language interoperability, that is, the ability of a certain tool to interact with external input–output formats. In other technological solutions like XML- or JSON-based languages, programmatic converters would be required. In contrast, DSMLs could be mapped at a higher level of abstraction by declaring and running transformation techniques (Anjorin et al. 2020; Burgueño et al. 2016; Brunelière et al. 2014). Model transformations are not only useful to create mappings between models, but also to increase or decrease the level of abstraction of artifacts (model or code). Specific transformation languages like QVT (Query/View/Transformation) by OMG (Gerpheide et al. 2016), ATL[1] (ATLAS transformation language) or ETL[2] (Epsilon transformation language) can be used to implement these mappings.

2.2 Data-Intensive Workflows

Pipelines in DI applications are multi-step processes where different tasks collaborate in order to meet a particular computationally intensive goal. These pipelines are usually formulated in terms of workflows. Workflows were conceived to be used in business and industrial contexts before their application to DI environments (WFMC 1999). Therefore, *business workflows* represent a common understanding of the business processes within organizations at a high level of abstraction, coordinating human activities and simple computing tasks. These types of process-centric workflows are designed to be automated, in a routine fashion, and according to well-established business rules that define the order in which tasks are executed. This is to say they are *control-driven* workflows, which means that they are focused on the control-flow perspective of business processes (vom Brocke et al. 2021).

In contrast, *DIWs* – or *scientific workflows* – that coordinate computationally intensive tasks, must deal with huge amounts of data and stringent performance requirements. Their execution order is implicitly derived from their data dependencies, i.e., they are typically *data-driven* (Curcin and Ghanem 2008), and must be fully automatable on an underlying computational infrastructure (Atkinson et al. 2017). As an example, Fig. 1 depicts a simple workflow for finding potential diseases from a set of input keywords. Rounded rectangles represent tasks, and square shapes represent data inputs and outputs (*R* stands for in-memory stored data and *F* for data stored in external files). Arrows represent data dependencies.

DIWs have been adopted in a large number of computational intensive areas, such as life science (Fillbrunn et al. 2017), astronomy (Ruiz 2014) or bioinformatics (Mullis et al. 2014). They have shown to provide numerous benefits in terms of reproducibility and validation. They also efficiently optimise the order and scheduling of executions, enable knowledge reuse, and improve data management, e.g., including data security or governance (Atkinson et al. 2017).

[1] ATL: https://www.eclipse.org/atl (last update: 18 Oct 2021; accessed 18 Mar 2022).

[2] ETL: https://www.eclipse.org/epsilon (last update: 6 Mar 2022; accessed 18 Mar 2022).

2.3 Data-Intensive Workflow Languages

DI applications have traditionally been implemented using general-purpose programming languages such as C, Java or Python, which now have extensions or libraries to support them. Other languages and frameworks such as R (Kohl 2015), Julia (Bezanson et al. 2012) or Swift (Zhao et al. 2007) have been developed to support scientific programming implementations. However, these languages require advanced programming skills, which prevents their wide use by domain experts with no specific computational knowledge.

WfMS try to mitigate the gap between programming and domain experts by providing functional elements, usually through graphical user interfaces that hide the difficulty of operating, optimising and managing computational resources. Traditionally, WfMS make use of their own language definitions, such as Triana[3] or LONI Pipeline,[4] oriented to scientific domains, and KNIME[5] (Konstanz Information Miner) to define data mining applications. However, these languages are tightly coupled to their corresponding WfMS and their specifications are not publicly open.

As an attempt to open their specifications, other WfMS have defined their workflow languages by means of XML schemas. SCUFL (Simple Conceptual Unified Flow Language) (Oinn et al. 2004) that was proposed by Taverna,[6] MoML (Modeling Markup Language) that was specifically designed for Kepler[7] and DAX (Directed Acyclic Graph in XML) that was developed for Pegasus,[8] are representative examples. While their specifications are publicly available in form of XML schemas, these languages are still intended to take advantage of the particular features provided by their WfMS. Some other platform-specific workflow languages were conceived to be executed on specific infrastructures. This is the case of GridAnt (Amin et al. 2004), which was proposed to describe pipelines for grid computing. Another example is the GridBus workflow enactment engine (Yu and Buyya 2009), which is based on xWFL, an XML-based language for the representation of quality of service requirements.

Two key limitations of these platform-specific workflow languages are: (1) their reduced interoperability with other workflow languages and execution engines, and (2) their limited reusability outside the platforms they were defined for (Deelman et al. 2009; de la Garza 2016). To address these problems, several workflow languages were defined in a more abstract and tool-independent manner. AGWL, IWIR and CWL are the best-known examples of these types of languages.

AGWL (Abstract Grid Workflow Language) (Fahringer et al. 2004) was the first attempt to create a platform-agnostic workflow language, even though it was originally created in the context of the tool ASKALON (Fahringer et al. 2007). Later, IWIR (Interoperable Workflow Intermediate Representation) (Plankensteiner et al. 2013) was designed to facilitate portability and interoperability between workflow-specific languages by decoupling the workflow logic from data and processors. More recently, CWL (Common Workflow Language) (Amstutz et al. 2020) is a language that can be executed on different software and hardware environments. CWL is based on JSON-LD (JSON for Linked Data).[9] Nevertheless, these approaches are still technology-dependent, as long as they depend on preset, parseable structures that conform to a concrete syntax based on JSON or XML. Dependence with these languages makes it difficult to capture the domain semantics, linking the expert to these technologies against changes in trends or language specifications (Brambilla et al. 2017). Table 1 shows a summary of the characteristics and limitations of each DIW-specific language.

General purpose business process modeling notations, such as SPEM,[10] BPMN,[11] CMMN,[12] or UML activity diagrams[13] could a priori be suitable candidates to specify DI processes. They possess the advantage that they are standard notations supported by many editors and other generic modeling tools such as MetaEdit+ (Kelly and Tolvanen 2021) or JetBrains MPS (Bucchiarone et al. 2021). However, they would require a major effort to implement extensions to describe the specific requirements and concepts of the types of DI processes and tasks used in the domain of interest. Moreover, effort would be required to cover other key aspects such as the target technological platforms or the data-driven execution engines that are not

[3] Triana:http://github.com/CSCSI/Triana (last update: 2014; accessed 10 Mar 2022).

[4] LONI Pipeline: https://pipeline.loni.usc.edu (last update: 2020; accessed 10 Mar 2022).

[5] KNIME: https://www.knime.com/ (last update: 2022; accessed 10 Mar 2022).

[6] Taverna: https://incubator.apache.org/projects/taverna.html (last update: 2020; accessed 10 Mar 2022).

[7] Kepler 2.5: https://kepler-project.org (last update: 2015; accessed 10 Mar 2022).

[8] Pegasus 5.0 https://pegasus.isi.edu (last update: 2020; accessed 10 Mar 2022).

[9] JSON-LD - JSON for Linked Data: https://json-ld.org/ (accessed 01 Mar 2022).

[10] Software & Systems Process Engineering Metamodel (SPEM), 2.0: https://www.omg.org/spec/SPEM/ (accessed 13 April 2023).

[11] Business process model and notation (BPMN), 2.0.2: https://www.omg.org/spec/BPMN/ (accessed 13 April 2023).

[12] Case management model and notation (CMMN), 1.1: https://www.omg.org/spec/CMMN/ (accessed 13 April 2023).

[13] Unified modeling language (UML), 2.5.1: https://www.omg.org/spec/UML/2.5.1/ (accessed 13 April 2023).

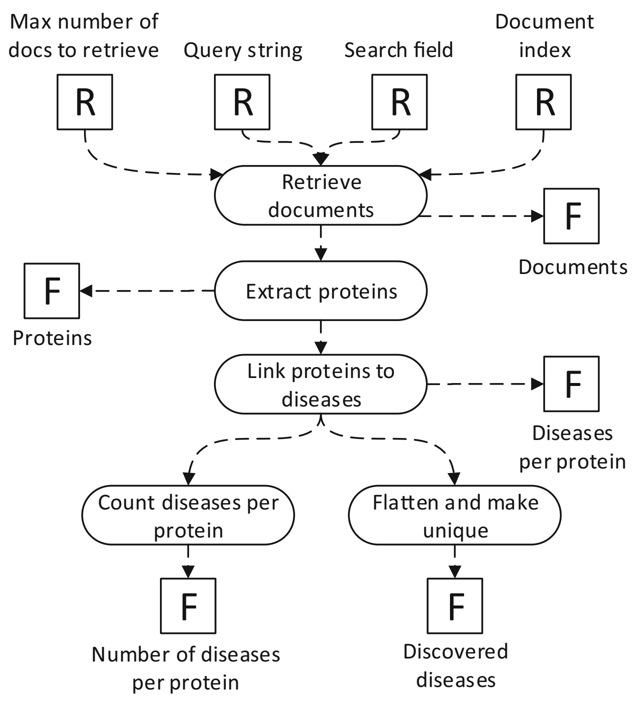


Fig. 1 Data-intensive workflow in bioinformatics [extracted from - Roure et al. (2008)]

supported by business process tools (Ludäscher et al. 2009). Our proposal uses a different approach because it defines generic metamodels that are syntax-agnostic, and could be used at the modeling level both to implement DIW notations and to exchange information between existing ones. At the execution level, our proposal can be used to specify the requirements of target platforms and workflow execution engines, and to connect to existing ones using model transformations.

3 Research Methodology

3.1 Design Science Research

The aim of this paper is the modeling of a DIW specification language (target primary artifact) according to the Design Science (Johannesson and Perjons 2014) methodological paradigm. Here, an artifact is an object created with the aim of solving a practical problem and can range from a definition, a model, a method or an instantiation in the form of a complete system. According to Hevner et al. (2004), DSR allows making research using design as a research method itself. In our problem, SWEL is the artifact to be designed and modeled and, thus, DSR allows ensuring that it is being built correctly (Dresch et al. 2015). Essentially, a key property of DSR is that it allows building our solution on exploiting specific problems, for example, by iteratively inspecting platform-specific languages or representation strategies.

The first activity of the DSR framework involves *explicating the problem*, as discussed in Sect. 3.2. Current efforts generate solutions that are not sufficiently generalizable and reusable. This is because e.g., (1) the rigidity of the platforms for the development of DIW-based applications; (2) their lack of interoperability; (3) the immobility of the knowledge generated; and (4) the high viscosity in a highly changeable context such as data science. Therefore, modeling an abstract language for high-level specification of DIWs becomes a primary design goal of our research. This solution should be independent of any tool or platform, thus enabling collaboration between data scientists by reusing fragments of cross-domain knowledge and interoperability of tools. Then, a second activity consists of outlining a solution to the problem by *defining requirements* for the solution artifact, as listed in Sect. 3.3.

Table 1 Summary of characteristics and limits of data-intensive workflow languages

Language	WfMS-coupled	Specification	Platform-specific	Notation-dependent
Triana	Yes	Close	No	Yes
LONI Pipeline	Yes	Close	No	Yes
KNIME	Yes	Close	No	Yes
SCUFL	Yes	Open	No	Yes
MoML	Yes	Open	No	Yes
DAX	Yes	Open	No	Yes
GridAnt	No	Close	Yes	Yes
GridBus	No	Close	Yes	Yes
xWFL	No	Close	Yes	Yes
AGWL	No	Open	No	Yes
IWIR	No	Open	No	Yes
CWL	No	Open	No	Yes
SWEL	No	Open	No	No

The next activity is the *design and development of the artifact*. In this paper, we create the SWEL language as the primary artifact, the design of which is explained in Sect. 4. The DSR framework then follows with the *demonstration of the developed artifact*, i.e., the activity where SWEL is used in a case study to show how it can solve a practical problem instance. Section 5 shows how SWEL supports the creation of tool support and Sect. 6 demonstrates its use in a case study.

The last activity in the cycle is to *evaluate the artifact*, so as to validate the extent to which the artifact solution solves the problem and satisfies the requirements (see Sect. 7.1). Based on the nature of the SWEL artifact, we have implemented an evaluation model grounded in continuous assessment. This approach has been consistently applied throughout the iterative refinement process, which has been informed by extensive feedback solicited from practitioners and experts. During the last iteration, a metrics-based evaluation is conducted in Sect. 7.2 to verify the suitability and adaptability of SWEL, using the evaluation framework outlined by Guizzardi et al. (2005). To validate the conclusions of specialists involved in the SWEL assessment during its development, a survey evaluation is performed in Sect. 7.3 with experts outside of the development activity.

3.2 Problem Motivation and Explanation

3.2.1 Problem Statement

The reuse of knowledge captured by DIWs across different tools with a similar purpose poses a challenge for experts in various domains (Sethi and Gil 2017; Garijo et al. 2017). This challenge was encountered first-hand by one of the authors at his company, a product-oriented company with over 25 million users. The company's data exploitation department created different DIWs using different tools at different times, depending on the problem at hand and their familiarity with the tool. The meaningful knowledge extracted is used to improve their services and products. Unfortunately, each DIW could only be executed by the tool for which it was created, and reusing workflow fragments required duplicating work. To address this issue, the department conducted an analysis to identify a single flexible tool for defining DIWs that could handle different types of DI problems. However, manually rewriting all DIWs in the new tool was not affordable in terms of time and effort. Moreover, concentrating knowledge on a single technology and requiring team members to learn a new tool was not desirable. Hence, a solution was needed to allow reuse and adaptation of knowledge captured by existing DIWs, regardless of the tool used. The idea of having a DSL that allowed this definition was only part of the solution to the problem; the other part involved the reuse of DIW knowledge across different underlying tools.

3.2.2 Identification of Core Elements

The DSML 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 DSML.

Initially, several meetings were conducted with different company data scientists. 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 (see Sect. 2.3). Finally, we relied on existing studies on common characteristics, frequently used execution models, and the typical tasks required for creating a DIW (see Sect. 3). 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 DAGs (Directed Acyclic Graphs) and those based on DCGs (Directed Cyclic Graphs). Both have traditionally been used to define DIWs, although DAG-based representations are commonly used in data-driven domains (Yu and Buyya 2006). 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 (see Sect. 2.3).

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.

3.2.3 *Identification of High-Level Tasks*

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, which are described in Sect. 2.3. We used these languages to identify the primary computational and visualization 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 analyzing their textual serialization, which is usually based on XML and JSON. We created several DIWs in different tools with the support of the company specialists and used a large number of DIWs from the public DIW repository myExperiment.org.

During the high-level task identification phase, we discovered that the core elements identified in the previous phase were applicable to the analyzed workflows. We also identified new elements that were not previously identified and incorporated them into SWEL as core elements.

3.2.4 *Identification of Extension Points*

We observed a wide variety of tasks within the different types of DIWs that were defined. In many cases, tasks depend directly on the characteristics of the tools for which the DIW was designed. A common example is tasks that invoke programs written in Java or Python and access data stored in external databases such as MySQL or MongoDB. It is not feasible to cover such a diverse range of tasks in a single language. However, by extending general elements, we can provide support for more specific elements such as those mentioned above, or those that depend on the platform where they run. These elements are identified as extension points, and they enable us to gather more specific and corporate knowledge to facilitate its reuse and unify terms.

3.2.5 *Identification of Elements Describing Scientific Experiments*

In collaboration with the company experts, we recognized the importance of including information about the project or scientific experiment that served as motivation for creating a DIW. This information provides a wider context for the design and objectives of the workflow, making it easier to comprehend the outcomes after execution and facilitate the transfer of knowledge between specialists.

This need arose while working with DIWs from repositories like myExperiment.org. Most of them incorporate this information in order to provide context for the workflows. However, it is worth noting that this information is not directly linked to the DIW, but 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 company. To that end, we searched for potential languages for defining scientific experiments in DI (Parejo 2013), with the goal of adapting a representation that is compatible with other proposals in SWEL.

3.3 Design Requirements

Taking the primary design objective to be met (Sect. 3.1), and in view of the detailed problem explanation (Sect. 3.2), the following requirements are defined for the solution artifact:

- REQ1: *To provide a suitable workflow language able to define DI applications at a high-level of abstraction.* The solution artifact must support the main features provided by current WfMS, such as cyclic and acyclic execution models, different workflow execution models (sequential, concurrent or iterative), data access methods like memory, database or external storage (Ferreira da Silva et al. 2017), data composition patterns (Montagnat et al. 2006), control flow structures (Curcin and Ghanem 2008), workflow composition, and data preparation, operation and visualization tasks (Garijo et al. 2014).
- REQ2: *To support the reuse of knowledge fragments between different tools by providing a platform-independent workflow language.* A platform-independent workflow language facilitates collaboration across multiple domains (Sethi and Gil 2017) and decouples the workflow definition from the lifecycle of a particular tool.
- REQ3: *To provide a notation-independent workflow language in terms of concrete syntax.* This independence requirement is aimed at enabling the best representation of domain semantics (Brambilla et al.

2017). Thus, the workflows defined by the solution artifact must be able to be represented in any notation, both textual and graphical, according to the domain requirements.

4 Design of SWEL

SWEL is an abstract, platform-independent DSML for the formulation of DIW. Its elements enable the definition of domain-specific concepts, constraints and interrelations conforming to the domain rules involved in the definition of DI pipelines. In this section, we first explain the overall structure of the language metamodel, which is organized in several packages. Then, each package is explained in detail.

4.1 Overall Structure of SWEL

A DIW contains a large set of diverse information artifacts, ranging from the computational execution specification to the domain concepts or to the project that led to its creation. The structure of the DIW determines the sequence of activities and dependencies between them as a DCG or DAG. This type of representation enables the definition of its execution efficiently by applying mechanisms such as parallelism or data composition. It hides the complexity about parallel or concurrent programming by providing high-level constructs to get the benefits of using such features. The structure of the DIW is defined according to the domain concepts and knowledge that data scientists consider relevant to meet the requirements of a particular scientific DI experiment (Garijo et al. 2014; Curcin and Ghanem 2008; Montagnat et al. 2006). Thus, the general organization of the architecture of SWEL is as follows:

- The *morphological layer* contains those elements that enable the low-level definition of a workflow. It is represented as a DCG or DAG, where its vertexes represent elements, and their arcs stand for dependencies between them.
- The *syntactic layer* consists of a set of packages enabling the representation of domain-specific requirements and workflow resources. At this level, elements allow the declaration of control structures, data types, fault-tolerant handlers, domain-specific tasks, and computational resources.
- The *specification layer* describes different information assets about the project, experiments and DI application associated to the definition of the workflow.

Specific details of these layers and its elements are omitted for space reasons, but the full specification is available as a technical report from the paper companion Website (Salado-Cid et al. 2023).

In Fig. 2, the overall structure of the language is represented as a UML package diagram, each package containing a specific part of the SWEL metamodel. In the following sections, the contents of each package are described. They are specified as UML class diagrams composed of metaclasses and their relationships. A metaclass defines a language element and may have attributes. Note that metaclasses are usually concrete, but they can be abstract too (its name depicted in italics), meaning that they cannot be directly instantiated to any language element. Extension points have also been defined to indicate those metamodel elements that are expected to be extended in the future, enabling the scalability of SWEL so that it can be adapted to different organizational and technological contexts.

4.2 Morphological Layer

The Morphological layer contains the elements representing the internal graph-based structure of a workflow, and consists of a single package, *ExecutionGraph*, depicted in Fig. 3. At this level, SWEL defines a workflow as a particular type of graph, namely an execution graph (metaclass *ExecutionGraph*), that is intended to define the set of execution steps needed to perform a computational process.

An execution graph is made up of computational operations (metaclass *Node*) and dependencies between them, represented as directed links (metaclass *DirectedEdge*). These operations, or nodes, are uniquely identified by a label and provide a set of connection points (metaclass *Endpoint*) to determine the sort of dependency to be established with the other nodes. On the one hand, a data dependency requires receiving or sending data from/to another node. Consequently, data connection points (*DataEndpoint*) are associated to the corresponding type of link, i.e., a data link (*DataLine*). On the other hand, there is a control dependency when a node can only be executed after another. This type of dependency is represented by control connection points (*ControlEndpoint*), and nodes are connected through control links (*ControlLine*). Moreover, the appearance of unexpected issues while running the workflow could trigger an alternative execution path with a different order between operations, namely an error dependency. Here, exception connection points (*ExceptionEndpoint*) are connected through an exception link (*ExceptionLine*). The definition of which specific connection point is associated to a specific link is performed by a linker (*Linker*).

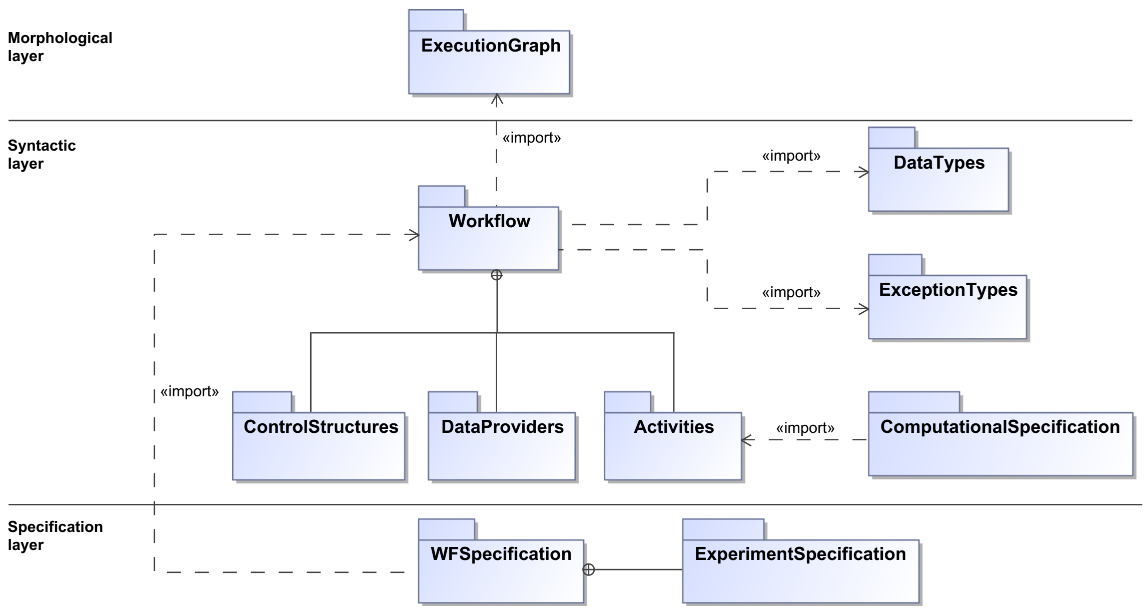
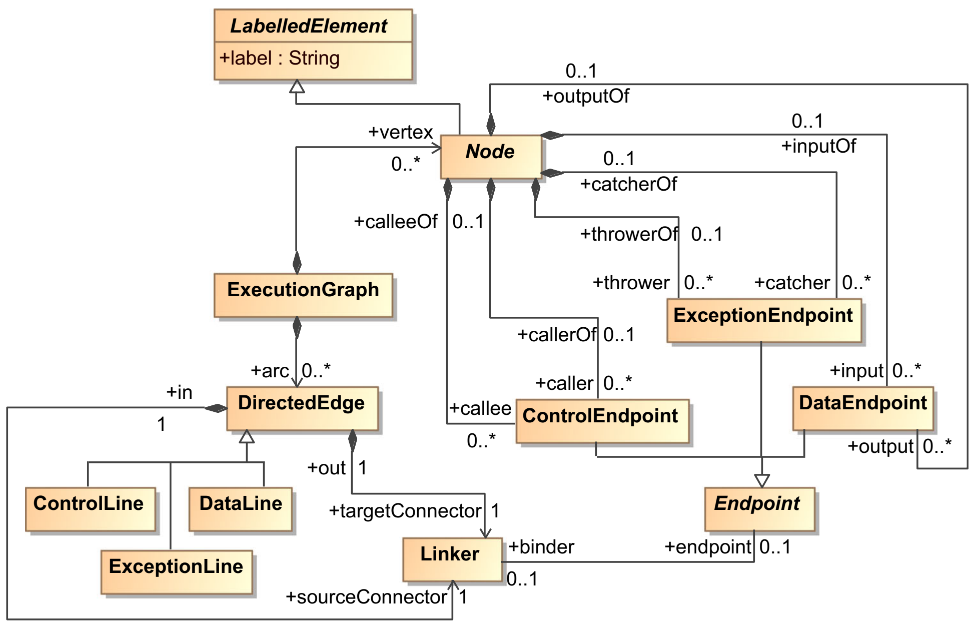


Fig. 2 Layers of the SWEL metamodel

Fig. 3 Metamodel elements within the Morphological layer: package *ExecutionGraph*



4.3 Syntactic Layer

The syntactic layer provides the elements that capture the knowledge extracted from domain experts. As shown in Fig. 2, this layer is composed of seven packages. The declaration of data and exception types is represented in packages *DataTypes* and *ExceptionTypes*. From the definition of the graph structure at the morphological layer, the package *Workflow* provides a set of adaptable building blocks to define pipelines. These consist of computational tasks (*Activity*), data providers (*DataProvider*) and control structures (*ControlStructure*), as well as dependencies between them. Finally, the package *ComputationalSpecification* contains the elements defining those specific computational resources that should be used in some particular scenarios, such as networks with dedicated servers, or cloud-based platforms. Note that this organization does not affect the language, but it provides a readable way to differentiate highly-coupled aspects of interest according to the design principle of separation of concerns. For brevity, we will focus only on those elements that are more relevant to the understanding of SWEL– see the technical report available at the paper companion Website for a full description of packages and meta-elements (Salado-Cid et al. 2023).

Regarding the packages *DataTypes* and *ExceptionTypes*, two sorts of data types (*DataType*) are defined in SWEL in order to classify the different data flows running through the pipeline. Basic data types are related to the primitive types of any programming language, such as integers, floats or strings. Complex data types define the format of a particular file, such as picture, audio or video. As for the exception elements (*ExceptionType*), they define alternative flows of execution (*ExceptionPath*) when an error happens, e.g., bad parameters, full disk, or permission denied. Note that an exception path is an exception endpoint, so exception elements are linked to particular nodes of the workflow. An error may have associated a list of actions (*Action*) to be performed. This could contain running a given task if the current activity fails, repeating its execution for several times, or stopping the pipeline execution. Both exception elements and actions can be extended to meet a wider range of values according to the particular needs of each scenario. For example, when a service is temporarily unreachable (*UnreachableCloudService*) in a cloud-based scenario and involves notifying the cloud supplier before finishing the workflow execution (*NotifyAndFinish*).

As shown in Fig. 4, the package *Workflow* declares a DIW (*Workflow*) as a set of computational assets (*Constructor*) and flows. Data flows focus on the handling of data whose availability implicitly determines the execution sequence (*DataDrivenWorkflow*). In certain cases, the execution flow must be explicitly defined, regardless of data availability (*ControlDrivenWorkflow*). Both approaches can also be combined (*ControlledDataDrivenWorkflow*). Having specific types enables the validation of the workflow to reduce human errors, such as adding a control structure to a data-driven workflow. Nevertheless, since the *Workflow* metaclass is not abstract, it could be directly instantiated, leaving the validity checking to the modeling tool implementing SWEL. Independently of the type of workflow, data flows are conveyed through data endpoints (*Port*) that define the type of data accepted by the computational asset.

In those cases where an explicit control of the pipeline execution is required, SWEL defines some control structures that are similar to those defined by BPMN or UML. The package *ControlStructures* provides different structures (*ControlStructure*) that enable the definition of both the starting point (*Begin*) and the ending point (*End*) of the workflow. Depending on the domain requirements, internally the execution flow can also be divided into parallel, concurrent flows (*Fork*), which allow some computational tasks to be executed simultaneously. Also, different execution paths can be joined either into a single flow that is executed each time that a joined flow is individually finished (*Merge*), or into a single flow executed only when all or part of the joined flows have ended (*Synchronizer*). Moreover, conditional structures (*Conditional*) facilitate the choice of the next asset or path to be executed. These conditions are decomposed into one or two operands (*Operand*) and an operator (*Operator*) that determines the type of condition to evaluate, which can be logical (*LogicalOperator*), relational (*RelationalOperator*) or mathematical (*MathematicalOperator*).

The definition of data-driven pipelines is the most common practice for DIW. As formalized in package *DataProviders*, data can be originated by different sources,

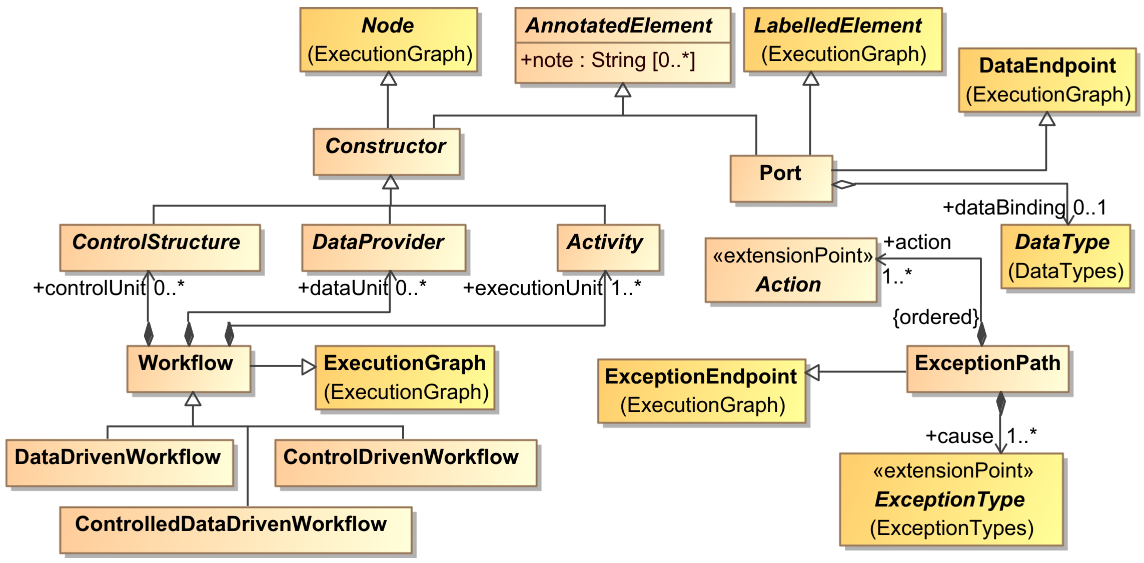
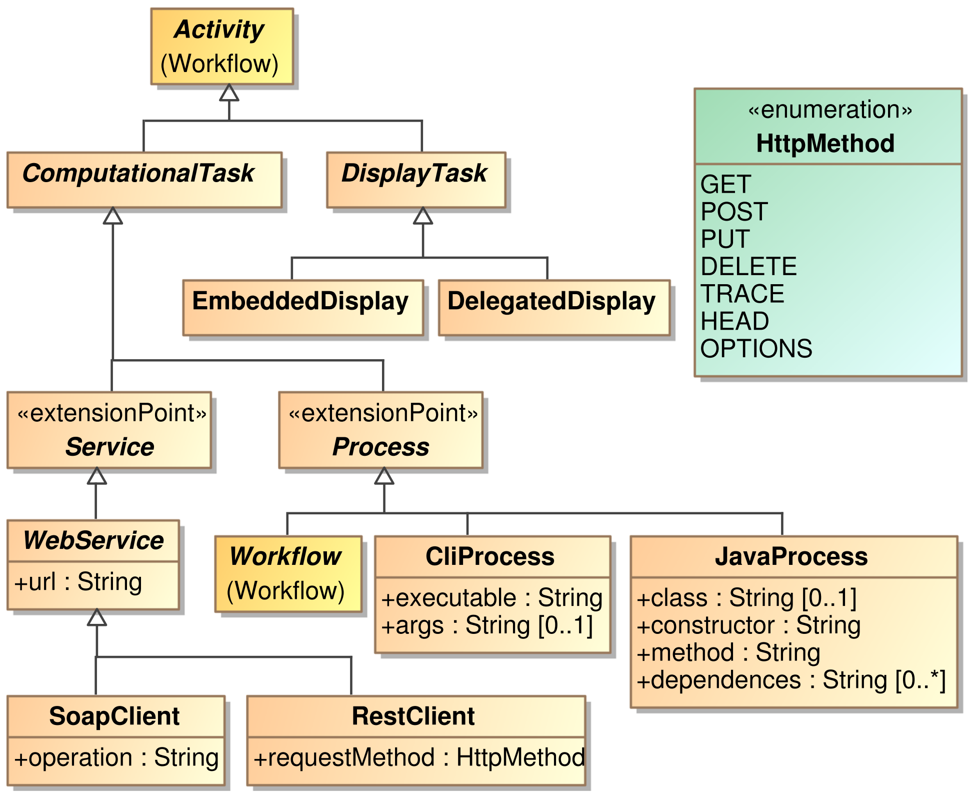


Fig. 4 Metamodel elements within the Syntactic layer: package *Workflow*

Fig. 5 Metamodel elements within the Syntactic layer: package *Activities*



to which the workflow accesses through providers (*Data-Provider*). Data providers enable the access to data depending on their location, e.g., in memory (*Record*), a file (*File*), a database (*Database*), or a data stream (*Stream*). Note that the analysis of data streams in DI applications, while possible, is not as common as that of batch data (Alaasam et al. 2021; Kranjc et al. 2015). The *Stream* element is designed to support the definition of data stream sources, the processing mode of which will depend specifically on the underlying workflow engine (Montagnat et al. 2006).

The extracted data is then transformed by tasks of different nature (see package *Activities* in Fig. 5). Computational tasks (*ComputationalTask*) are executable tasks performed without human interaction to transform inputs into new output data. These tasks consist of either invoking services (*Service*) – e.g., Web services (*WebService*) including REST[14] (*RestClient*) and SOAP[15] (*SoapClient*) – or processes (*Process*), such as Java processes (*JavaProcess*) or a command-line interface program (*CliProcess*). Note that both *Service* and *Process* are extension points. This means that other types of tasks can be added to SWEL if needed. For example, some tasks involving humans might be needed in a few specific DI domains, but this is not usual. Moreover, workflows can also be considered as processes themselves in those cases where the definition of nested workflows is considered.

Another type of activity refers to data visualization tasks (*DisplayTask*), which facilitate the readability and understanding by the human being of data and other results. Two main types of visualization tasks have been defined: those that are integrated and configured within the pipeline (*EmbeddedDisplay*); and those that show the result by invoking some external visualisation tool (*DelegatedDisplay*).

Finally, a few particular requirements on the pipeline (e.g., security issues or platform-specific constraints) may require an explicit specification of the execution environment and computational resources. Usually, this information can be used for a particular WfMS to leverage the underlying execution platform or specify the necessary computational requirements. In package *ComputationalSpecification* (see Fig. 6), these computational resources (*ComputationalResource*) allow specifying both the host location (*Host*), constraints on the platform and operating system (*OperatingSystem*), and CPU and GPU requirements (elements *CPU* and *GPU*, respectively). Note that the definition of these computational resources is extensible to define particular platform-specific

[14] REST - REpresentational State Transfer: https://www.w3.org/2001/sw/wiki/REST (accessed 23 Feb 2022).

[15] SOAP Specifications: https://www.w3.org/TR/soap/ (accessed 23 Feb 2022).

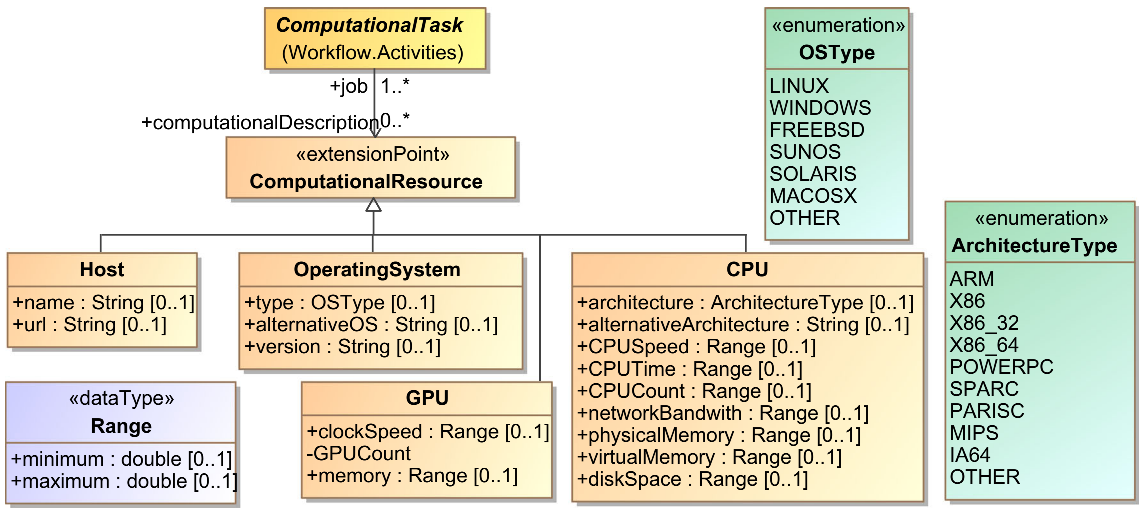


Fig. 6 Metamodel elements within the Syntactic layer: package *ComputationalSpecification*

computational resources and compatible in its current form with JSDL (Job Submission Description Language).[16]

4.4 Specification Layer

Raising the abstraction level, the package *WFSpecification* (see Fig. 7a) declares the elements related to the meta-information about the workflow. For example, a general description, its terms and conditions of use, or the version number. This is non-executable information that provides a general description of the project (*Project*), such as its name, license, or version. Information about the stakeholders (*Stakeholder*) includes the description of the organization (*Organization*) and participants (*Person*).

Furthermore, DIWs are usually designed in the context of an experimentation or project, which is metamodeled by package *ExperimentSpecification* (see Fig. 7b). SWEL allows providing meta-information about a DI experiment (*Experiment*), which can be formulated to validate or reject a hypothesis (*Hypothesis*). This is frequently a computational experiment, which is configured by a set of properties (*Configuration*). Since there are different types of scientific experiments and it is not within the scope of SWEL to cover all of them, only an essential type of experiment is provided (*BasicExperiment*). Nevertheless, the inclusion of new types is allowed through the extension point. Note that the concepts of this layer rely on those already used by other languages specialized in the definition of scientific experiments, such as SEDL (Scientific Experiments Description Language) (Parejo 2013).

[16] Job submission description language (JSDL) specification, 1.0: http://www.ogf.org/documents/GFD.136.pdf (accessed 18 April 2023).

5 Tool Support

The formalisation of the SWEL metamodel and the specification of concrete notations facilitate the development of MDE-based tools. Even though SWEL is platform- and notation-independent, graphical and textual syntaxes are essential for making DIWs accessible to application domain experts and workflow tools. This section presents two examples of DIW specification and execution structures using graphical and JSON textual syntaxes, respectively. Interested readers can download these notations and tools from the paper companion Website (Salado-Cid et al. 2023).

5.1 Model-Based WfMS for Multiple DI Domains

User requirements led to the implementation of a WfMS to facilitate DIW creation and execution across various domains. The tool features domain-agnostic components that can be customized for domain-specific workflows. As depicted in Fig. 8, its architecture consists of a user-centric graphical editor and a workflow engine, both of which are highly customizable. The editor helps domain experts define and represent domain-specific DI applications, while the workflow engine executes workflow activities using available computational resources in an effective way.

The graphical editor is a design- and end-user-oriented environment that employs a SWEL-concrete syntax to represent and monitor workflows. As an illustrative example, Table 2 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 metaclass to be mapped to a particular *Graphical element*. An example data-driven workflow taken from a public repository (Roure et al. 2008) using the graphical notation is shown in Fig. 9. This workflow calculates the number of publications and citations per year for a specific author using a biomedical information service. The workflow includes searching for publications, extracting citations and publication years, and displaying the results using 2D histograms and a report. The workflow engine uses data dependencies and input/output ports to determine the execution order and data composition patterns.

The graphical user interface provides data scientists with the necessary elements to draw and configure a DIW, including (1) a palette of available workflow components, (2) a canvas area to insert and connect components and their dependencies, and (3) a design assistance tool. The graphical editor allows direct verification of conformance between the concrete syntax and the language metamodel. Here, users can work at different levels of abstraction for the same workflow definition. The outputs will be presented by the data visualizer either graphically or in textual form, depending on the configuration of the display components. In addition, the workflow repository manages storage, retrieval, exportation, and importation of DIWs. The workflow execution manager enables management and monitoring of the workflow execution, gathering execution traces and data shown to users in the graphical editor.

The workflow execution engine interprets and executes the activities defined in the workflow. The engine provides all the features required for the invocation of services, local execution of computing programs, and data management. The engine consists of a scheduler, executor, and monitoring module. The scheduler analyses the high-level workflow definition and generates the corresponding low-level executable specification. This module coordinates and optimizes the execution considering the computational constraints and execution workload. The executor module runs the corresponding computing programs or services. It can integrate external tools and platforms for grid and distributed computing, increasing the computational capabilities and reducing the execution time. Finally, the monitoring module logs the execution to provide information related to time, available memory, and outcomes to monitor how resources are managed and consumed, and how resulting data are generated.

5.2 JSON Concrete Syntax Validator

It is also possible to use SWEL as a concrete textual notation for the serialization of DIW models. JSON[17] is an IETF (Internet Engineering Task Force) standard language widely used as a data exchange format on the Web via, e.g., REST APIs and services. Listing 1 shows a snippet of the workflow represented graphically in Fig. 9, but using a JSON-based textual notation for SWEL. This syntax defines project meta-information (*Specification* package in Sect. 4.4) and DIWs in the second level, which include nodes (type, attributes, configuration, notes, etc.) and the control or data links between them. In the example, the JSON code 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). The available complete JSON notation scheme (Salado-Cid et al. 2023) enables the development of SWEL-based data exchange and Web services.

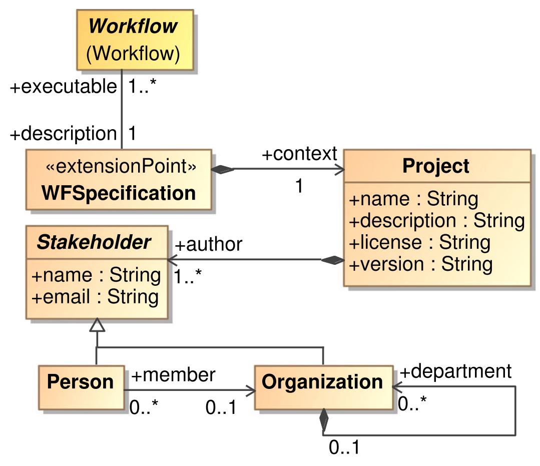
Listing 1: Code snippet of a JSON-based notation for SWEL

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

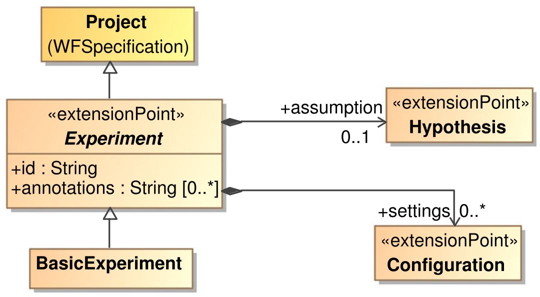
6 Demonstration of SWEL

To demonstrate the applicability of the artifacts generated and the modeling framework, we have used SWEL to explore its suitability as a mechanism for the interoperability between WfMSs. Knowledge in workflows

[17] The JavaScript Object Notation (JSON) Data Interchange Format (No. IETF RFC 8259). https://datatracker.ietf.org/doc/html/rfc7159.



(a) Package *WorkflowSpecification*.



(b) Package *ExperimentSpecification*.

Fig. 7 Metamodel elements within the Specification layer

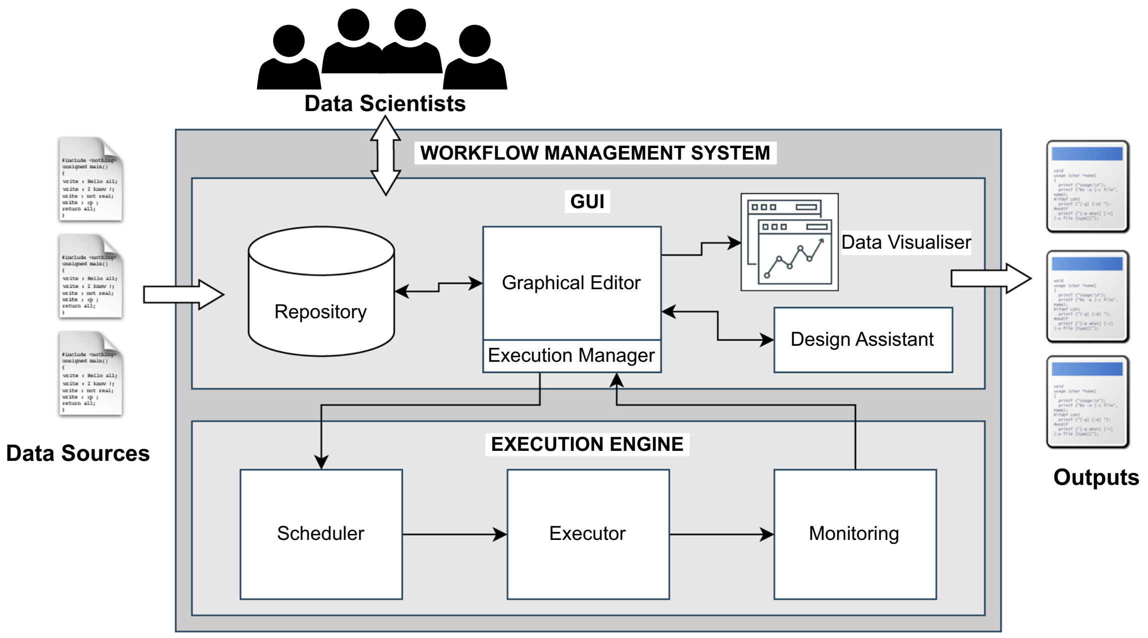


Fig. 8 Architecture of the model-based WfMS

frequently needs to be reused in or adapted to different application domains, or even under the same domain but having been created with different tools. For example, we could assume a data scientist generating a pipeline using the Taverna tool. If a fragment of that workflow needs to be reused or shared for use in another WfMS such as Kepler, it should be rendered again in the new tool. Here, in addition to the costs, resources and time required, factors such as the modeler interpretation and potential inaccuracies in the representation of the original pipeline come into play. As discussed in Sect. 2.1, interoperability is one of the intended benefits with the use of MDE techniques. More specifically, the application of the horseshoe model (Kazman et al. 1998) allows the reuse of pieces of content generated using different tools, see Fig. 10. Workflows are defined at different levels of abstraction in order to obtain models of a source system artifact, to transform those models into some target models, and to finally generate the new system artifact.

Table 2 Partial examples of concrete syntax [extended in the supplementary material Salado-Cid et al. (2023)]

SWEL type	SWEL element	Graphical element
DirectedEdge	DataLine	⇢
	ExceptionLine	→
Activity	ComputationalTask	■
	DisplayTask	□
DataProvider	Record	⚙
	File	📄
Experiment	BasicExperiment	⬚

In this case study, SWEL can serve as the pivot element of the horseshoe process between the platform-specific workflow languages SCUFL and MoML. Figure 10 shows the implemented process. Starting from the source code generated by Taverna in the SCUFL language, its representation is extracted as a SCUFL model, in conformance with the SCUFL metamodel. The abstract representation of the user-generated pipeline is then converted into a SWEL model by applying one-way model transformations. This platform-agnostic SWEL model specification is subsequently transformed into MoML models (Kepler) and text code. Since there are no formalized metamodels for SCUFL and MoML, we have defined their corresponding platform-specific metamodels, partially gathering the main features required for showing this interoperability case study. Note that the horseshoe model shown in Fig. 10 could be extended by incorporating other platform-specific metamodels according to the specific interoperability needs. In this case, new branches coming from or targeting to another WfMS could be added.

To illustrate this particular case, Fig. 11 shows a workflow in SCUFL to extract information about a particular gene from a nuclear protein database. We want to

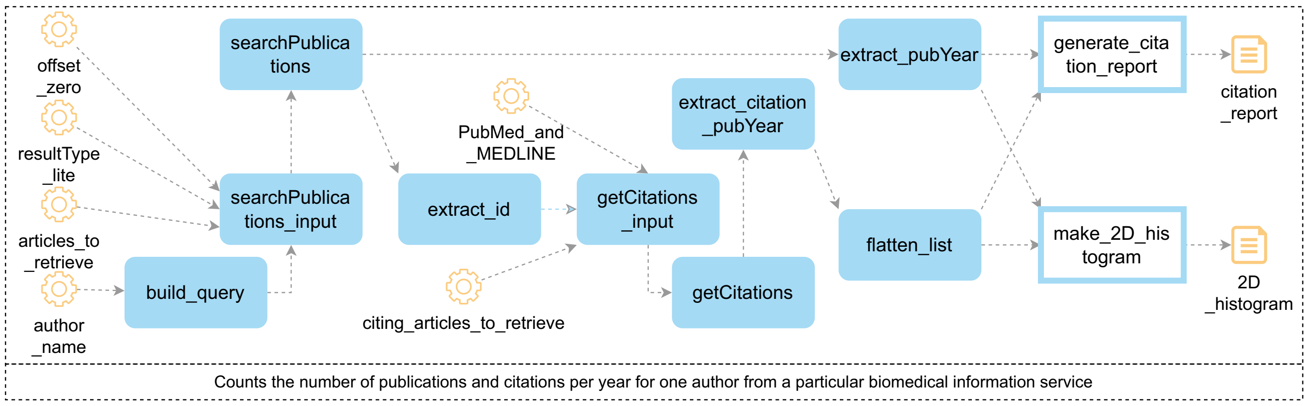


Fig. 9 SWEL representation of a data-driven workflow

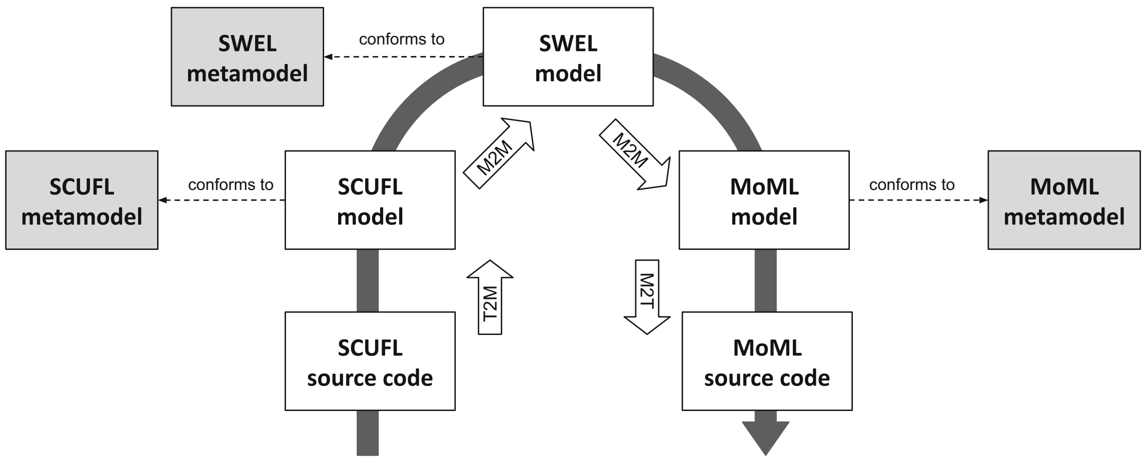


Fig. 10 Horseshoe process using SWEL to achieve interoperability

convert it into its corresponding workflow in MOML, see Fig. 12. For this we follow the horseshoe process, as shown in Fig. 10.

First, a text-to-model (T2M) transformation is defined to transform the SCUFL source code into its model representation. Since SCUFL is a XML-based language, the standard language XSLT (eXtensible Stylesheet Language for Transformations) is used. For example, Listing 2 shows the XSLT template that transforms SCUFL data links (*Datalink*) into their corresponding model representations. Similar templates are used to transform the rest of the SCUFL elements and structures such as workflows, data inputs and outputs, and processors.

links (*Datalink*) into data lines (*Dataline*), and input and output endpoints (*Endpoint*) into input and output ports (*Port*). As an example, the *MapRetryDispatchLayer* QVT relation is depicted in Listing 3, mapping the dispatch layer when retrying an execution after an error detection (*ConfigBeans::RetryConfig*) from SCUFL (lines 4–6) into the corresponding exception path in SWEL (*Workflow::ExceptionPath*) (lines 7–11).

Listing 2: Datalinks template definition

```xml
<xsl:template name="datalinks" match="datalinks/datalink">
  <connection><source>
    <xsl:attribute name="type"><xsl:value-of select="source/@type" /></xsl:attribute>
    <xsl:attribute name="port"><xsl:value-of select="source/port" /></xsl:attribute>
    <xsl:attribute name="processor"><xsl:value-of select="source/processor" /></xsl:attribute>
  </source><target>
    <xsl:attribute name="type"><xsl:value-of select="sink/@type" /></xsl:attribute>
    <xsl:attribute name="port"><xsl:value-of select="sink/port" /></xsl:attribute>
    <xsl:attribute name="processor"><xsl:value-of select="sink/processor" /></xsl:attribute>
  </target></connection>
</xsl:template>
```

Listing 3: QVT relation from SCUFL to SWEL of the *MapRetryDispatchLayer* declaration

```
 1 relation MapRetryDispatchLayer {
 2     ttimes : Integer;
 3     tdelay : Integer;
 4     checkonly domain source s : SCUFL::ConfigBeans::RetryConfig
 5     {   maxRetries = ttimes,
 6         initialDelay = tdelay };
 7     enforce domain target t : SWEL::Workflow::ExceptionPath
 8     {   action = retry : SWEL::Workflow::Retry {
 9         {   times = ttimes,
10             delay = tdelay },
11         cause = unknown : SWEL::ExceptionTypes::UnknownException {} }; }
```

Then, model transformations declared in QVT define the relations and dependencies between the SCUFL and the SWEL models. Each relation is executed when its preconditions are met, and defines a set of post-conditions to determine the execution order of subsequent relations. A first relation initiates the conversion into a SWEL workflow (*Workflow*) by transforming each SCUFL element into SWEL, e.g., processors (*Processor*) into nodes (*Node*), data

The current workflow definition in SWEL is expected to be transformable into any other metamodeled workflow representation. Thus, following the horseshoe process, new model-to-model (M2M) transformations are defined in QVT to declare the mapping between the SWEL and MoML model elements. An initial transformation converts each SWEL workflow node and line into the corresponding MoML entity (*Entity*), relation (*Relation*) and link (*Link*). The QVT *MapRecord* relation is implemented in Listing 4,

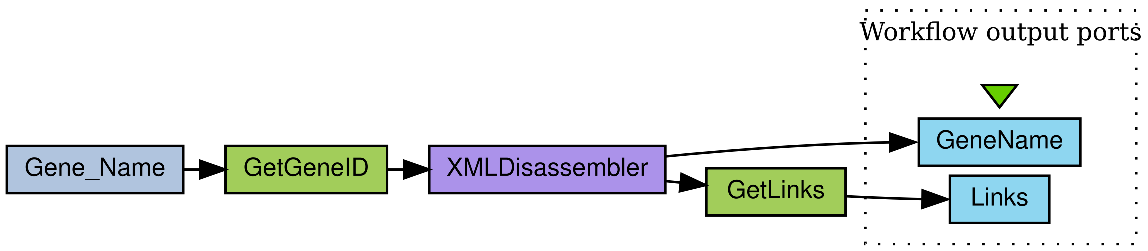


Fig. 11 The source SCUFL workflow representation

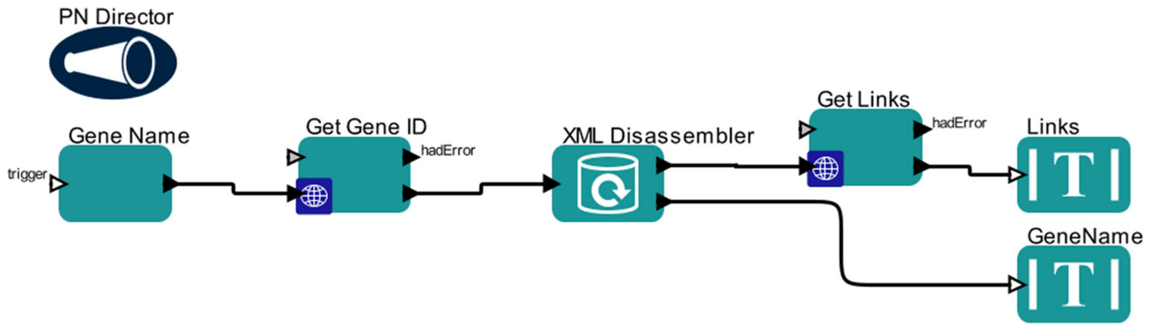


Fig. 12 The Kepler-specific MoML workflow representation

which shows the transformation of a record in SWEL (*Workflow::DataProviders::Record*) (lines 3–4) into the corresponding MoML model element (*Entity*) (lines 5–18).

Listing 4: QVT relation from SWEL to MoML of the *MapRecord*

```
1  relation MapRecord {
2      evalue : String;
3      checkonly domain source r : SWEL::Workflow::DataProviders::Record
4      { value = evalue };
5      enforce domain target e : MoML::Entity
6      {   class = 'ptolemy.actor.lib.StringConst',
7          attribute = value : MoML::Property
8          {   name = 'value',
9              class = 'ptolemy.data.expr.Parameter',
10             value = evalue },
11         attribute = firingCountLimit : MoML::Property
12         {   name = 'firingCountLimit',
13             class = 'ptolemy.data.expr.Parameter',
14             value = 'NONE' },
15         attribute = none : MoML::Property
16         {   name = 'NONE',
17             class = 'ptolemy.data.expr.Parameter',
18             value = '0' } };
```

From this platform-specific MoML model, model-to-text (M2T) transformations are declared in Acceleo[18] to generate the Kepler source code. Acceleo uses the MOF2T (MOF model-to-text transformation language) standard.[19] A first template initiates the generation of all elements by invoking the *GenerateEntity* template (see Listing 5). Here, MoML entities and their subentities, with a given configuration and their connections, are created. The

[18] Open-source template-based code generator Acceleo: http://www.eclipse.org/acceleo (accessed 18 Mar 2022).

[19] MOF model to text transformation language (MOFM2T), 1.0: http://www.omg.org/spec/MOFM2T/1.0/ (accessed 13 April 2023).

resulting workflow is finally depicted in Fig. 12 following the Kepler notation.

Listing 5: Acceleo template of the *GenerateEntity* declaration

```
[template public generateEntity(entity : Entity)]
<entity name="[entity.name/]" class="[entity.class/]">
    [for (property : Property | entity.attribute)]
        [generateProperty(property)/]
    [/for]
    [for (subentity : Entity | entity.subentity)]
        [generateEntity(subentity)/]
    [/for]
    [for (port : Port | entity.endpoint)]
        ...
        <port name="[port.name/]" class="[port.class/]">
            ...
        </port>
        ...
    [for (relation : Relation | entity.connection)]
        <relation name="[relation.name/]" class="[relation.class/]"
            ...
        </relation>
    [/for]
    [for (link : Link | entity.binder)]
        [generateLink(link)/]
    [/for]
</entity>
[/template]
```

Note that all these model transformations between the different languages are defined at the language level, and not at the concrete application level. This means that they only need to be defined once for each pair of languages, and then simply executed for each application written in the source language. In addition, the definition of the model transformations has served to iterate and refine the SWEL model to cover the main features of other existing languages. On the other hand, the lack of standardization in defining DIWs can lead to incompatible functionalities in existing WfMSs, which can hinder the intended interoperability.

We have collected the definition of the metamodels, the transformations used in this section, together with the tools that have been developed to illustrate the entire transformation process. These, alongside a few videos illustrating their usage are available at the SWEL companion Website (Salado-Cid et al. 2023).

7 Evaluation of SWEL

7.1 Requirements Validation

In line with the DSR methodology, this section shows the extent to which the artifact solution solves the problem and satisfies the requirements described in Sect. 3. In order to validate these requirements, a solution artifact, SWEL, and multiple supporting products, which are considered as prototypes, are generated (see Sects. 5 and 6). *Prototyping* is a common and well-known validation technique, which allows testing and experimenting with the presented model to check if it meets their specified requirements. Another technique used has been continuous *requirements review*, where requirements are reviewed and contrasted to check for any errors and ambiguity. Table 3 represents the main research requirements, as well as the generated outputs that have been used for validation. Both the primary and support artifacts generated [see the companion Website (Salado-Cid et al. 2023)] allow to cover all the research requirements set for their validation. Two of the authors of this paper are experts in the area of data science, and they were the first ones to use and test all developed artifacts. Furthermore, both the language and the associated tools were continuously evaluated by the specialists from the company where the problem was identified, to check that the results helped solve their original problems. They

Table 3 Outcomes for research requirements validation

	REQ1	REQ2	REQ3
SWEL:Morphological layer (Sect. 4.2)	X		
SWEL:Syntactic layer (Sect. 4.3)	X		
SWEL:Specification layer (Sect. 4.4)	X		
Concrete syntax for SWEL: graphical notation (Sect. 5.1)	X		X
Concrete syntax for SWEL: JSON-based notation (Sect. 5.2)	X		X
JSON concrete syntax validator (Sect. 5.2)			X
SWEL WfMS: Graphical editor (Sect. 5.1)			X
SWEL WfMS: Workflow repository (Sect. 5.1)		X	
SWEL WfMS: Workflow execution engine (Sect. 5.1)		X	
Case study: T2M and M2T (Sect. 6)		X	
Case study: M2M (Sect. 6)		X	
Interoperability tool: SCUFL, MoML, CWL (Sect. 6)		X	

REQ1: *High-level workflow language for DI applications;* ***REQ2***: *Knowledge reuse among tools with platform-independent workflow language;* ***REQ3***: *Notation-independent workflow language in terms of concrete syntax.*

provided valuable feedback during the whole process, ensuring that the results were useful and accurate according to their needs.

7.2 Quantitative Evaluation

In order to evaluate the suitability and adaptability of SWEL when defining DIWs, a quantitative analysis has been conducted, using the framework proposed by Guizzardi et al. (2005) as a reference. This framework evaluates modeling languages based on four comparative metrics: lucidity, soundness, laconicity and completeness. We have conducted the comparison of SWEL against three DIW languages, which we have considered to represent the most important ones: SCUFL and MoML are the languages defined by two popular workflow tools (Taverna and Kepler, respectively); CWL is a recent approach aimed at establishing an open standard. As such, note that the evaluation results should be interpreted in this context, particularly with regard to the completeness, expressiveness and conciseness of the language. Only those elements of SWEL that are used in data-driven workflows have been taken into account in order to make a fair comparison. Furthermore, since SWEL is an extensible language, some extension points of SWEL have been defined to support specific elements of the compared languages. Particularly, a list of extension points has been defined in SWEL to meet all the required particular tasks required by SCUFL: *Process$SCUFLBeanShell* defines those activities implemented using BeanShell[20] (a Java compatible scripting language); *Process$SCUFLRshell* defines those activities implemented using R programming language; *Process$SCUFLInteraction* helps humans to define some basic case-specific inputs; *Process$SCUFLXPath* defines expressions to query or transform XML documents; *Process$SpreadsheetImport* enables the read of spreadsheet-like data; and *Process$SCUFLLocalworker* that defines the information about Java programs. Moreover, an extension point to define JavaScript code has been implemented for CWL (*Process$CWLExpressionTool*).

[20] BeanShell: https://beanshell.github.io/ (accessed 24 Sept 2022).

Table 4 SWEL abstract syntax quantitative evaluation

Metrics	SCUFL	MoML	CWL
Lucidity	13/18 (72.22%)	11/12 (91.67%)	3/13 (23.08%)
Soundness	13/18 (72.22%)	11/12 (91.67%)	7/13 (53.85%)
Laconicity	13/13 (100%)	11/11 (100%)	5/7 (71.43%)
Completeness	13/13 (100%)	11/11 (100%)	7/7 (100%)

The results of these metrics are shown in Table 4. *Lucidity* measures the degree of clarity of SWEL in terms of how many elements of the language have a unique representation in the other languages. The values obtained show that SWEL is highly expressive, concise and clear. In the case of CWL, the low percentages are mainly because this language is still in the definition phase and provides few elements to specify DIWs. This fact also influences the calculation of *soundness*, which determines the degree of correspondence of SWEL elements with elements of other languages. *Laconicity* measures how concise our language is by considering the number of elements from other languages that correspond to each element in SWEL, and its resulting values are close to the maximum in all cases. Finally, *completeness* indicates the degree to which SWEL is compatible with the other languages. Note that this metric is crucial when determining the suitability of SWEL to achieve interoperability between WfMS, and it is the measure in which SWEL achieves the highest score. The concrete mappings between the elements of SWEL and the rest of the languages used to compute these metrics are listed in Table 5. Note that these mappings can be one-to-

Table 5 Defined mappings between data-driven elements of SWEL and the languages SCUFL, MoML and CWL

SWEL	SCUFL	MoML	CWL
SoapClient	Activity [class="net.sf.taverna.t2.activities.wsdl.WSDLActivity"]	Entity [class= "org.sdm.spa.WSWithComplexTypes"]	-
RestClient	Activity [class="net.sf.taverna.t2.activities.rest.RESTActivity"]	Entity [class= "org.kepler.actor.rest.RESTService'"]	-
CliProcess	Activity[class="net.sf.taverna.t2.activities.externaltool. ExternalToolActivity"]	Entity [class= "ptolemy.actor.lib.Exec"]	CommandLineTool
JavaProcess	–	Entity	–
EmbeddedDisplay	–	Entity[class="ptolemy.actor.lib.gui.Display"]	–
Workflow	Workflow		Workflow
Port	Port	Port	(1) WorkflowStepInputParameter (2) WorkflowStepOutputParameter (3) InputToolParameter (4) OutputToolParameter
Record	Activity[class="net.sf.taverna.t2.activities.stringconstant. StringConstantActivity"]	Entity[class="ptolemy.actor.lib.StringConst"]	–
File	–	Entity[class="org.geon.BinaryFileReader"]	–
Stream	–	–	–
Database	–	Entity[class="org.geon.DatabaseQuery"]	–
Dataline	Datalink	Relation	(1) WorkflowStepInputParameter (2) OutputToolParameter
Process$SCUFLBeanShell	Activity[class="net.sf.taverna.t2.activities.beanshell. BeanshellActivity"]	N/A	N/A
Process$SCUFLRshell	Activity[class="net.sf.taverna.t2.activities.rshell.RshellActivity"]	N/A	N/A
Process$SCUFLInteraction	Activity[class="net.sf.taverna.t2.activities.interaction. InteractionActivity"]	N/A	N/A
Process$SCUFLXPath	Activity[class="net.sf.taverna.t2.activities.xpath.XPathActivity"]	N/A	N/A
Process$SCUFLSpreadsheetImport	Activity[class="net.sf.taverna.t2.activities.spreadsheet. SpreadsheetImportActivity"]	N/A	N/A
Process$SCUFLLocalworker	Activity[class="net.sf.taverna.t2.activities.localworker. LocalworkerActivity"]	N/A	N/A
Process$CWLExpressionTool	N/A	N/A	ExpressionTool

Language elements not covered by other languages are indicated by a dash. "N/A" implies that the element should not be considered in the calculation of the different metrics, since it is based on an extended element one (1:1), one-to-many (1:N) when one SWEL element can be mapped to several elements of the target language. Alternatively, these mappings can be many-to-one (N:1) when several SWEL elements can be mapped to the same target language element.

7.3 Expert Evaluation

As detailed in Sect. 3.1, SWEL has undergone iterative refinement through an incremental validation process, using extensive feedback from practitioners and experts in data science and DIWs. In this section, we present a human-based survey evaluation to assess the adequacy of SWEL as an intermediate model in the interoperability process between the Taverna and Kepler tools. We also analyze the suitability and comprehensibility of SWEL as a representation language for DIWs. For this purpose, a team of eleven experts, not involved in the prior design, development, and evaluation of SWEL, was selected. Five of the experts work in academia in diverse areas of data science, while six are data scientists in industry. Both senior and junior profiles have been considered in both cases. These external experts have extensive knowledge of DI applications and were, therefore, well-suited to provide additional insights. Note that these specialists may not necessarily be experts in the DI tools that are the subject of this experiment, although they are familiar with the use of WfMSs.

To complete the experiment, the experts used the transformation tool presented in the case study in Sect. 6 and responded to a questionnaire on common criteria in tool evaluation (Mijac 2019). The criteria chosen to formulate the questions referred to the efficacy, usefulness, accuracy, effectiveness, validity, and completeness of the workflow conversion process between Taverna and Kepler. They also analyzed the practicality and comprehensibility of the SWEL model generated by the tool. The package containing the Q &A of the experiment is available from the companion Website (Salado-Cid et al. 2023).

The experiment consisted of three exercises: (1) a first training transformation on a given workflow; (2) a transformation of a workflow chosen by the expert and downloaded from a public repository; and (3) an analysis of the adequacy and the comprehensibility of SWEL as a representation model for DIWs, as well as its accuracy in incorporating the concepts of the source (Taverna) and target (Kepler) models. Each expert responded to a total of 18 questions, rated on a scale of 1 (lowest) to 10 (highest).

According to the survey results, the experts gave exercise 2 (workflow transformation) an average valuation of 9.38. Therefore, we consider that the tool solves the problem for which it has been formulated. We can also conclude that the models by which it was inspired, including the partial models extracted from SCUFL and MoML, as well as those from SWEL, are suitable in this particular scenario. With regard to exercise 3, it should be noted that they are not experts in MDE. As a result, it was challenging for them to differentiate the unfriendly XML notation necessary for serialization and reading[21] from the abstraction of the metamodel. Nevertheless, the experts gave an average valuation of 7.86 for comprehensibility and practicability of the morphological level compared to 8.68 for the syntactic level. This difference is consistent with the level of abstraction of the information represented. In fact, we speculate that the morphological level was expected to score lower as it is usually transparent to the data scientist, and the workflow enactment subsystems handle it. Finally, an average valuation of 8.73 supports the relevance of including meta-information in the DIW itself. Both the questionnaire and the disaggregated data obtained from the survey experiment can be found at the companion Website (Salado-Cid et al. 2023).

7.4 Threats to Validity

According to Wohlin et al. (2012), there are four basic types of validity threats that can affect the validity of our study. We cover each of these in the following.

External Validity These threats are related to the extent to which it is possible to generalize the findings and conclusions of this study. First, the comparison evaluation has been conducted with a selected set of DIW languages, which we have considered to represent the most important ones. However, there might be others, or new ones may appear, that may challenge our results especially regarding the conciseness, completeness and expressiveness of SWEL. Its extension points were defined precisely to address this issue, but we cannot foresee all the features that might appear in the future. Anyway, SWEL could evolve as new languages or important features appear. Second, the specialists from the industry who confirmed the validity of SWEL during its design and development were employed within the same corporation, which may result in biased views towards their needs and application domain. To validate practicability of SWEL as an interoperability tool, and the suitability and comprehensibility of the proposed metamodel, a survey experiment was conducted with other eleven experts from up to five different corporations, as well as academia. However, further experiments with a larger sample of users and industry specialists are planned as future work.

Internal Validity These threats are related to the factors that could affect the results of our evaluation. All

[21] Intermediate SWEL models were generated using the standard format XMI (XML Metadata Interchange): http://www.omg.org/spec/XMI/ (accessed 18 April 2023).

developed tools and software artifacts (metamodels, model transformations, etc.) have been double-checked for correctness and consistency to mitigate these threats. However, we would have to conduct further experiments to reconfirm such claims. Also, to ensure that the semantics of DIWs are preserved by the model transformations when converting them across languages, this would be validated formally. In practice, this is complex and would require an extensive research work in a different direction.

Construct Validity These threats are concerned with the relationship between theory and what is observed, and are related with those issues that might arise during research design. We have used a comparative framework between our proposal and other DIW approaches. However, there are two aspects that may pose a threat to the construct validity. On the one hand, to the best of our knowledge, SWEL is the only proposal formalized as a metamodel, but it is being compared against non-metamodeled proposals. For this purpose, we have partially metamodeled some current technologies (SCUFL, MoML, CWL), using reverse re-engineering. However, such metamodels might not be accurate or complete. So far, our experiments confirm that they are appropriate and complete, but further validations can be performed by conducting more interoperability experiments with all types of DIW applications, which is planned future work. Moreover, SWEL is platform-independent, so it does not focus on those features specifically offered by any given tool. Again, the SWEL extension mechanisms have been designed precisely to address this issue. We think that these extensions will be sufficient to cover all necessary features, but it may be the case that a new feature or a certain property of a language cannot be expressed with them. If this were the case, we would have to evolve the language to take them into account.

Conclusion Validity These threats are concerned with the issues that affect the ability to draw correct conclusions and whether the results can be repeated. First, to deal with this threat we have made publicly available all the artifacts developed and used in this work. Secondly, further experiments with diverse external users and industry specialists can be carried out to evaluate SWEL. This would confirm its properties in other environments, as well as validating the suitability of SWEL in different situations.

8 Concluding Remarks

This paper presents SWEL, a DSML that provides an extensible metamodel for the specification of scientific workflows at different levels of abstraction, from the high-level specification of the DI problem to the low-level representation of the connected, executable graph. In addition to the metamodel, its constraints and governing rules, SWEL can be mapped to different concrete notations, both textual and graphical, allowing its adaptability to diverse organizational contexts and tools. Both a JSON-based validator and a graphical editor have been developed using our proposal.

To the best of our knowledge, this is the only formally metamodeled proposal so far. This offers a powerful mechanism for defining model transformations that leverage the interoperability and adaptability of knowledge assets. To this end, in this paper we have also validated the proposal by presenting an exemplary application, a quantitative metric-based evaluation of SWEL against other related proposals, as well as a survey evaluation with external experts. The results show that, compared to other languages, SWEL is a language suitable for defining DIWs and enabling interoperability between tools. In addition, the surveyed external experts have supported the benefits in terms of comprehensibility and practicability that brings the layered metamodel of SWEL.

We believe that MDE can live a second youth with the expansion of no-code and low-code applications. In this direction, we intend to explore SWEL as a future avenue of research, to offer no-code technological solutions that are interoperable and can be synchronized with other technology-dependent tools. In addition, MDE and SWEL could facilitate the automated creation of domain-specific tools, reusing imported knowledge assets from multiple sources and repositories. Regarding our proposal, we plan to conduct more experiments and validate SWEL with additional case studies and the development of more applications. This will include applications that require deployment of specific technological ecosystems such as cloud or grid platforms. This would expand the evaluation of its extensiveness, expressiveness and interoperability capabilities. We also plan to improve the SWEL toolkit, including its usability by industrial practitioners for developing DI applications, or the formal analysis of its internal components by, e.g., checking that the model transformations used by SWEL preserve the semantics of the application semantics. Finally, we want to empirically evaluate the usability of our proposal through further experiments with more users and industry specialists, in order to find possible improvements to our language that can help broaden its use and value for the DI applications community.

9 Additional Material

For the sake of transparency and replicability, we have made available all the artifacts mentioned in the paper at the SWEL companion Website (Salado-Cid et al. 2023): a technical report with the formal specification of the SWEL metamodel; the developed tools demonstrating its use, namely the JSON validator and the graphical editor; the validation artifacts described in Sect. 6, including the metamodels extracted from SCUFL, MoML and CWL and the model transformations used for the implementations of the tools; and the responses from experts in the survey experiment.

Acknowledgements Partially supported by the Spanish Ministry of Science, grant PID2020-115832GB-I00 funded by MICIN/AEI/10.13039/50110 0011033, and by the Spanish Government (FEDER/Ministerio de Ciencia e Innovación-Agencia Estatal de Investigación) under projects PID2021-125527NB-I00 and TED2021-130523B-I00.

Funding Funding for open access publishing: Universidad de Córdoba/CBUA.

References

Alaasam ABA, Radchenko GI, Tchernykh AN (2021) Micro-workflows data stream processing model for industrial internet of things. Supercomput Front Inn 8(1):82–98

Altintas I, Berkley C, Jaeger E, Jones M, Ludäscher B, Mock S (2004) Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings of the international conference on scientific and statistical database management (SSDBM), vol 16, pp 423–424

Amin K, von Laszewski G, Hategan M, Zaluzec N, Hampton S, Rossi A (2004) GridAnt: a client-controllable grid workflow system. In: Proceedings of HICSS'04

Amstutz P, Crusoe MR, Tijanić N, Chapman B, Chilton J, Heuer M, Kartashov A, Leehr D, Ménager H, Nedeljkovich M, Scales M, Soiland-Reyes S, Stojanovic L (2020) Common workflow language description, v1.2. https://w3id.org/cwl/v1.2/

Anjorin A, Buchmann T, Westfechtel B, Diskin Z, Ko HS, Eramo R, Hinkel G, Samimi-Dehkordi L, Zündorf A (2020) Benchmarking bidirectional transformations: theory, implementation, application, and assessment. Softw Syst Model 19(3):647–691

Atkinson M, Gesing S, Montagnat J, Taylor I (2017) Scientific workflows: past, present and future. Futur Gener Comput Syst 75:216–227

Bezanson J, Karpinski S, Shah VB, Edelman A (2012) Julia: a fast dynamic language for technical computing. CoRR arXiv: 1209.5145

Boubeta-Puig J, Ortiz G, Medina-Bulo I (2015) ModeL4CEP: graphical domain-specific modeling languages for CEP domains and event patterns. Expert Syst Appl 42(21):8095–8110

Brambilla M, Cabot J, Wimmer M (2017) Model driven software engineering in practice, 2nd edn. Morgan and Claypool, Williston

Brunelière H, Cabot J, Dupé G, Madiot F (2014) Modisco: a model driven reverse engineering framework. Inf Syst Technol 56(8):1012–1032

Bucchiarone A, Cicchetti A, Ciccozzi F, Pierantonio A (2021) Domain-specific languages in practice: with JetBrains MPS. Springer International, New York

Buhl HU, Röglinger M, Moser F, Heidemann J (2013) Big data. Bus Inf Syst Eng 5(2):65–69

Burgueño L, Wimmer M, Vallecillo A (2016) A linda-based platform for the parallel execution of out-place model transformations. Inf Syst Technol 79:17–35

Campos C, Grangel R (2018) A domain-specific modelling language for corporate social responsibility (CSR). Comput Ind 97:97–110

Chen CP, Zhang CY (2014) Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci 275(Supplement C):314–347

Coleman T, Casanova H, Pottier L, Kaushik M, Deelman E, Ferreira da Silva R (2022) WfCommons: a framework for enabling scientific workflow research and development. Futur Gener Comput Syst 128:16–27

Curcin V, Ghanem M (2008) Scientific workflow systems—can one size fit all? In: 2008 Cairo international biomedical engineering conference (CIBEC'08), pp 1–9

de la Garza L et al (2016) From the desktop to the grid: scalable bioinformatics via workflow conversion. BMC Bioinform 17:127

Deelman E, Gannon D, Shields M, Taylor I (2009) Workflows and e-science: an overview of workflow system features and capabilities. Futur Gener Comput Syst 25(5):528–540

Deelman E et al (2015) Pegasus, a workflow management system for science automation. Futur Gener Comput Syst 46:17–35

Demchenko Y, Grosso P, de Laat C, Membrey P (2013) Addressing big data issues in scientific data infrastructure. In: Proceedings of CTS'13. IEEE, pp 48–55

Dresch A, Lacerda D, Valle Antunes Jr JA (2015) Design science research: a method for science and technology advancement. Springer, Cham

Fahringer T, Pllana S, Villazon A (2004) AGWL: abstract grid workflow language. In: International conference on computational science. Springer, Heidelberg, pp 42–49

Fahringer T, Prodan R, Duan R, Hofer J, Nadeem F, Nerieri F, Podlipnig S, Qin J, Siddiqui M, Truong HL, Villazon A, Wieczorek M (2007) ASKALON: a development and grid computing environment for scientific workflows. Springer, Heidelberg, pp 450–471

Ferreira da Silva R, Filgueira R, Pietri I, Jiang M, Sakellariou R, Deelman E (2017) A characterization of workflow management systems for extreme-scale applications. Futur Gener Comput Syst 75:228–238

Fillbrunn A, Dietz C, Pfeuffer J, Rahn R, Landrum GA, Berthold MR (2017) Knime for reproducible cross-domain analysis of life science data. J Biotechnol 261:149–156

Fowler M (2010) Domain specific languages, 1st edn. Addison-Wesley, London

Garijo D, Alper P, Belhajjame K, Corcho O, Gil Y, Goble C (2014) Common motifs in scientific workflows: an empirical analysis. Futur Gener Comput Syst 36:338–351

Garijo D, Gil Y, Corcho O (2017) Abstract, link, publish, exploit: an end to end framework for workflow sharing. Futur Gener Comput Syst 75:271–283

Gerpheide CM, Schiffelers RRH, Serebrenik A (2016) Assessing and improving quality of QVTo model transformations. Softw Qual J 24(3):797–834

Guizzardi G, Ferreira Pires L, van Sinderen M (2005) An ontology-based approach for evaluating the domain appropriateness and comprehensibility appropriateness of modeling languages. In: Proceedings of MODELS'05. Springer, Heidelberg, pp 691–705

Hamdaqa M, Met LAP, Qasse I (2022) iContractML 2.0: a domain-specific language for modeling and deploying smart contracts onto multiple blockchain platforms. Inf Softw Technol 144

Hevner AR, March ST, Park J, Ram S (2004) Design science in information systems research. MIS Q 28:75–105

Johannesson P, Perjons E (2014) An introduction to design science. Springer, Cham

Kazman R, Woods SS, Carrière SJ (1998) Requirements for integrating software architecture and reengineering models: CORUM II. In: Proceedings of WCRE'98, pp 154–163

Kelly S, Tolvanen J (2021) Collaborative modelling and metamodelling with MetaEdit+. In: Proceedings of MODELS'21 companion. IEEE, pp 27–34

Kohl M (2015) Introduction to statistical data analysis with R. Ventus Publishing ApS, London

Kranjc J, Smailovic J, Podpecan V, Grcar M, Znidarsic M, Lavrac N (2015) Active learning for sentiment analysis on data streams: methodology and workflow implementation in the clowdflows platform. Inf Process Manag 51(2):187–203

Ludäscher B, Weske M, McPhillips T, Bowers S (2009) Scientific workflows: business as usual? Springer, Heidelberg, pp 31–47

Mijac M (2019) Evaluation of design science instantiation artifacts in software engineering research. In: Proceedings of CECIIS'19. Springer, Heidelberg, pp 313–321

Montagnat J, Glatard T, Lingrand D (2006) Data composition patterns in service-based workflows. In: 2006 workshop on workflows in support of large-scale science (WORKS'06), pp 1–10

Mullis T, Liu M, Kalyanaraman A, Vaughan J, Tague C, Adam J (2014) Design and implementation of Kepler workflows for BioEarth. Procedia Comput Sci 29:1722–1732

Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, Carver T, Glover K, Pocock M, Wipat A, Li P (2004) Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics 20(17):3045–3054

Parejo JA (2013) Moses: a metaheuristic optimization software ecosystem. applications to the automated analysis of software product lines and service-based applications. Ph.D. thesis, University of Sevilla, Sevilla

Plankensteiner K, Prodan R, Janetschek M, Fahringer T, Montagnat J, Rogers D, Harvey I, Taylor I, Balaskó Á, Kacsuk P (2013) Fine-grain interoperability of scientific workflows in distributed computing infrastructures. J Grid Comput 11(3):429–455

Roure DD, Goble C, Bhagat J, Cruickshank D, Goderis A, Michaelides D, Newman D (2008) myExperiment: defining the social virtual research environment. In: 4th IEEE international conference on e-science. IEEE Press, pp 182–189

Ruiz J et al (2014) Astrotaverna—building workflows with virtual observatory services. Astron Comput 7–8:3–11

Salado-Cid R, Ramírez A, Romero JR (2018) On the need of opening the big data landscape to everyone: challenges and new trends. Springer, Heidelberg, pp 675–687

Salado-Cid R, Vallecillo A, Munir K, Romero JR (2023) SWEL companion website. https://doi.org/10.5281/zenodo.8085894

Schlauderer S, Overhage S (2018) BoSDL: an approach to describe the business logic of software services in domain-specific terms. Bus Inf Syst Eng 60(5):393–413

Sethi RJ, Gil Y (2017) Scientific workflows in data analysis: bridging expertise across multiple domains. Futur Generat Comput Syst 75:256–270

Szalay A, Gray J (2006) Science in an exponential world. Nature 440(2020 Computing):413–414

Tera Allas JB, Chui M, Dahlström P, Hazan E, Henke N, Ramaswamy S, Trench M (2018) Artificial intelligence is getting ready for business, but are businesses ready for AI? In: Analytics comes of age, McKinsey Analytics, pp 18–34

van der Aalst W, Damiani E (2015) Processes meet big data: connecting data science with process science. IEEE Transact Serv Comput 8(6):810–819

vom Brocke J, Baier MS, Schmiedel T, Stelzl K, Röglinger M, Wehking C (2021) Context-aware business process management. Bus Inf Syst Eng 63(5):533–550

WFMC (1999) Terminology & glossary. Technical Report, WFMC-TC-1011, Workflow Management Coalition

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2012) Experimentation in software engineering. Springer, Heidelberg

Yu J, Buyya R (2006) A taxonomy of workflow management systems for grid computing. J Grid Comput 3(3):171–200

Yu J, Buyya R (2009) Gridbus workflow enactment engine. CRC Press, Cambridge, pp 119–146

Zhao Y, Hategan M, Clifford B, Foster IT, von Laszewski G, Nefedova V, Raicu I, Stef-Praun T, Wilde M (2007) Swift: fast, reliable, loosely coupled parallel computation. In: Proceedings of SCW'07. IEEE Computer Society, pp 199–206