Towards Machine-actionable FAIR Digital Objects with a Typing Model that Enables Operations

Maximilian Inckmann
Scientific Computing Center
Karlsruhe Institute of Technology
Karlsruhe, Germany
ORCiD: 0009-0005-2800-4833

Nicolas Blumenröhr
Scientific Computing Center
Karlsruhe Institute of Technology
Karlsruhe, Germany
ORCiD: 0009-0007-0235-4995

Rossella Aversa
Scientific Computing Center
Karlsruhe Institute of Technology
Karlsruhe, Germany
ORCiD: 0000-0003-2534-0063

***Abstract*—FAIR Digital Objects support research data management aligned with the FAIR principles. They provide a machine-actionable and harmonized metadata layer on top of existing systems and specifications. However, the typing models used in current Data Type Registries lack mechanisms for inheritance and to associate operations with data types. We build on the existing concepts and present a typing model that enables reusable, technology-agnostic FAIR Digital Object Operations associated with data types. The Integrated Data Type and Operations Registry with Inheritance System is a prototype of our typing model that supports data type inheritance, rule-based validation, and the computation of type-operation associations. We conclude that this work substantially advances the machine-actionability of FAIR Digital Objects, paving the way towards dynamic, interoperable, and reproducible research workflows.**

***Index Terms*—Data Type Registry, FAIR Digital Objects, FAIR Digital Object Operations, Machine-actionability, Metadata**

I. Introduction

The rapid advancement in digital technologies has significantly transformed scientific research, facilitating the collection, processing, and analysis of extensive datasets. However, the growing diversity and complexity of research data present substantial challenges, particularly in terms of findability, accessibility, interoperability, and reusability [1]. The FAIR guiding principles [2] were established to address these issues and provide best practices for sustainable data management.

FAIR Digital Objects (FAIR-DOs) [3], [4] embody these principles by providing a machine-actionable, persistent, and harmonized representation of (meta)data beyond the borders of data spaces on top of existing infrastructures [1], [5]. Rather than storing the actual research artifacts, FAIR-DOs offer well-described pointers to the systems where those artifacts reside. FAIR-DOs rely on globally unique, resolvable, and persistent identifiers (PIDs) and their persistent records, typically managed by the well-established Handle system [6]–[8], which ensures their longevity and reliability. Every value inside a FAIR-DO record is typed using a PID-referenced data type that defines its syntax and semantics. Reusing these data types, wherever their syntax and semantics fit, ensures that identical references have identical meaning. Profiles aggregate multiple data types to define the structure of FAIR-DO records. Domain-agnostic profiles, such as the Helmholtz Kernel Information Profile [9], are used to harmonize essential information in FAIR-DOs. A strong and manifested type system of FAIR-DOs is therefore the foundation for machines to automatically interact with them and with their referenced resources across research domains [5], [10].

To enable machine-actionability, FAIR-DOs must support a mechanism for type-based interaction with their contents and the resources they refer to. These FAIR-DO Operations must be modeled in a typed, interoperable, technology-agnostic, and reusable manner. Operations and their dependencies must be described agnostic to execution environments with their technological differences. Associating operations with data types leads to a highly interconnected typing model that needs to be managed, queried, and validated.

Existing Data Type Registries (DTRs) [11] with their typing models represent a significant development towards machine-interpretable FAIR-DOs. Their schema-based architectures lack support for complex relationships between entities, making them unsuitable for type-associated FAIR-DO Operations and inheritance mechanisms for data types.

To address these gaps, we introduce a typing model for a new graph-based FAIR-DO type system and its prototypical implementation, the Integrated Data Type and Operations Registry with Inheritance System (IDORIS). IDORIS supports inheritance, type-associated FAIR-DO Operations, and mechanisms capable of validating and processing the complex relationships of our model. Our developments are based on the core concepts and lessons learned from existing DTR systems. This work contributes to FAIR (research) data management by advancing the machine-actionability of FAIR-DOs.

In Section II, we provide a brief insight into existing DTRs and current work on FAIR-DO Operations. We introduce our integrated typing model for data types, FAIR-DO Operations, and a type-operation association mechanism in Section III. IDORIS is described in Section IV and is evaluated together with the model in Section V based on a use case. Finally, we summarize our key contributions and discuss future research directions in Section VI.

This project is funded by the Helmholtz Metadata Collaboration Platform (HMC) and supported by the consortium NFDI-MatWerk, funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the National Research Data Infrastructure – NFDI 38/1 – project number 460247524.

II. Related Work

FAIR-DOs are a data management approach related to Linked Data [12], RO-Crate signposting [13], and nanopublications [14]. This work focuses on Handle-based FAIR-DOs due to their persistence, strong typing, and validated, machine-interpretable content [5], [11], [15]. Every FAIR-DO is described by an information record of key-value pairs, persistently stored in a federated Handle Registry[1], and resolvable by a Handle PID [5]–[8]. A key in the information record uses a PID to reference a machine-interpretable information type in a DTR. This allows the value to be validated against the referenced information type, making it machine-interpretable.

EIDR [16] is a widely adopted harmonization layer for the entertainment industry that structurally resembles FAIR-DOs via DOI-identified key-value records. Unlike FAIR-DOs, EIDR uses a fixed schema instead of a type system, does not store information within their Handle records, and is a centralized registry [17]. Similarly, other technologies, such as OWL [18] with SHACL [19] and signposting with RO-Crates [13], do not meet the properties of Handle-based FAIR-DOs. Therefore, their adoption is not further explored in the scope of this paper.

Within the Research Data Alliance (RDA)[2], multiple working groups and interest groups established outcomes and recommendations on FAIR-DO content [20], information typing models [21], and DTRs [22]. This work has been acknowledged, among others, by the European Commission [1], [23]. International initiatives such as the RDA and the FAIR Digital Objects Forum[3] provide a valuable platform for discussion, collaboration, early implementations, and adoption of FAIR-DOs. Comprehensive practical solutions addressing critical gaps in the FAIR-DO typing infrastructure remain an open challenge.

The existing typing infrastructure comprises three Data Type Registry instances – the ePIC test DTR[4], the ePIC production DTR[5] and the EOSC DTR[6]. Those DTRs follow the same typing model, based on: *PID-BasicInfoTypes*, *PID-InfoTypes* and *KernelInformationProfiles* [11], [20]. However, their implementations differ slightly between DTR instances and are not standardized. These DTRs have supported several FAIR-DOs use cases in multiple domains such as material sciences [24], digital humanities [25], and energy research [26]. LLMs have been shown to effectively interpret Handle-based FAIR-DOs [27].

Technologically, all current DTRs are based on Cordra [28], a JSON schema-based metadata repository that is only able to validate syntactic compliance with JSON schemas. Thus, the primary focus of existing DTRs has been limited to syntactic validation, lacking support for semantic validation and associating executable operations to specific data types. Moreover, despite the recognized benefits of object-oriented programming (OOP) principles, such as inheritance and polymorphism, current DTRs do not support these mechanisms. This lack of sophisticated logic significantly restricts the semantic richness and operational flexibility needed to represent complex, interconnected data resources commonly encountered in scientific research. In addition, existing DTRs do not support systematic reuse of type definitions, significantly hindering scalability and maintainability. We envision the creation of Kernel Information Profiles (KIP) [20] using curated domain-specific attributes in addition to those already provided by domain-agnostic profiles such as the Helmholtz KIP [9]. Existing DTRs lack support for reuse, forcing redundant remodeling of domain-agnostic profiles by creators of "extending" domain-specific profiles. This redundant work is grossly inefficient, error-prone, and a missed opportunity to leverage the de facto subtyping relationship between domain-specific and domain-agnostic KIPs.

Conceptually, FAIR-DO Operations provide a mechanism for interacting with FAIR-DOs, the values contained within the Handle records, and the external resources they reference [10], [29]. Existing approaches for service-oriented FAIR-DO Operations typically focus on basic CRUD (Create, Read, Update, Delete) functionalities as specified by the Digital Object Interface Protocol (DOIP) [30]. These operations target entire FAIR-DOs and must be individually implemented in each service that supports such operations [30]. This enables a perfectly machine-actionable API, but is limited to services actively supporting DOIP, defeating the technology-agnosticity of FAIR-DOs. DOIP seems promising as a FAIR-DO oriented communication protocol beyond service-oriented Operations because it also allows for more complex, type-specific operations, and specifies a pattern for operation discovery and execution.

Currently, there is no method to describe technology-agnostic FAIR-DO Operations independently of specific executing services. Furthermore, mechanisms to dynamically associate operations with FAIR-DOs according to at least one FAIR-DO association mechanism, such as "Record typing", "Profile typing", and "Attribute typing", as described in [29], still need to be developed. These existing limitations highlight the need for a more advanced typing infrastructure that is capable of supporting sophisticated semantic validation, inheritance management, polymorphism, and robust type-associated FAIR-DO Operations within FAIR-DO ecosystems.

III. Typing Model

We introduce a typing model for FAIR-DOs, which describes technology-agnostic operations and their association with data types. For better readability, we draw analogies to principles from object-oriented programming (OOP). We will use the term "typing" to refer to the information available within FAIR-DO records and our typing model, similar to the "information typing" used within current data transfer records (DTRs). This contrasts with the "FAIR-DO typing" described in [29], which evaluates the availability of operations.

[1]https://handle.net
[2]https://rd-alliance.org
[3]https://fairdo.org/
[4]https://dtr-test.pidconsortium.net
[5]https://dtr-pit.pidconsortium.net
[6]https://typeregistry.lab.pidconsortium.net

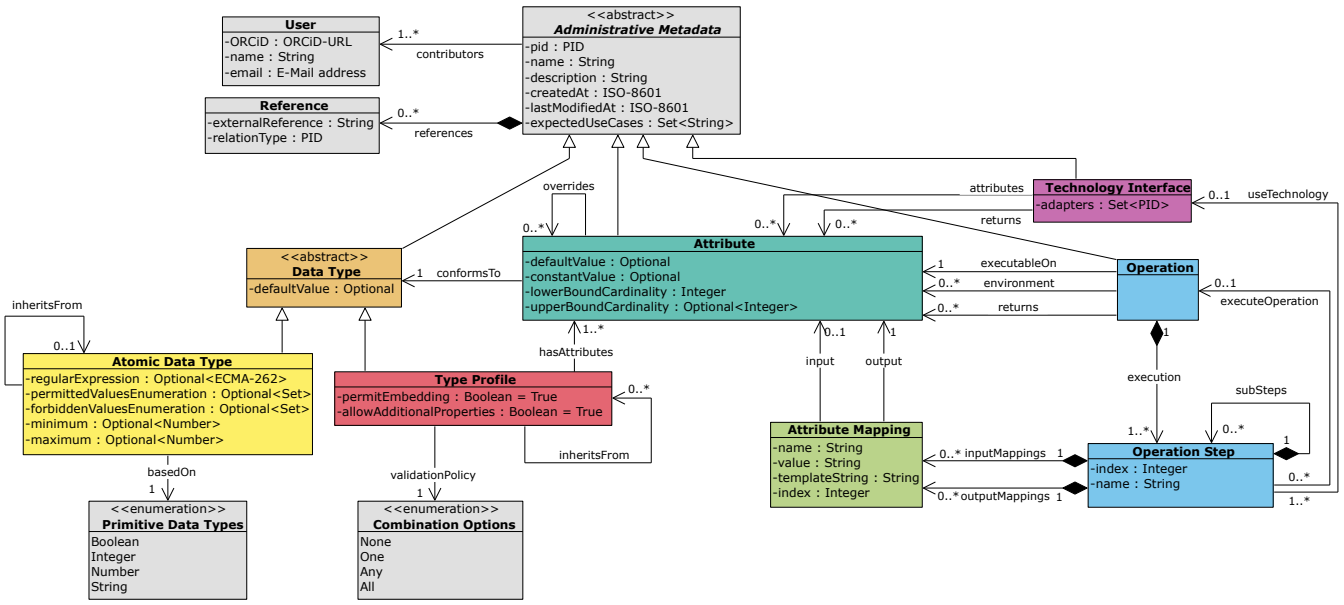

Fig. 1. UML model class diagram of the complete typing model

Figure 1 illustrates our typing model in a UML diagram, including *Data Type* (orange), *Atomic Data Type* (yellow), *Type Profile* (red), and *Attribute* (dark green). FAIR-DO Operations are associated with our typing model through the *Attribute* class and consist of instances of the classes *Operation* and *Operation Step* (blue), *Technology Interface* (purple) and *Attribute Mapping* (light green). Gray elements represent enumerations and implementation-dependent metadata. We will italicize the class names in lowercase for instances and in uppercase for classes.

A. Data Types

We use "Data Type" as a generalized term to refer to "Atomic Data Types" and "Type Profiles" by specifying the *Data Type* class as an abstract superclass of the *Atomic Data Type* and *Type Profile* classes. This abstraction allows us to reference *data types* uniformly and therefore reduces redundancy and improves semantic clarity. It also decouples the attribute logic from specific data types (Section III-B).

1) Atomic Data Types: Atomic data types define the syntax of values in FAIR-DO records. They are built on top of JSON primitives (boolean, integer, number, and string) to enable serialization. Therefore, *atomic data types* are comparable to primitive data types in OOP, but offer additional restriction mechanisms that enable a more rigorous validation of values: For any *atomic data type*, enumerations of permitted and forbidden values can be specified, which take precedence over the following mechanisms. Strings can be limited by specifying a regular expression as well as a minimum and maximum length. Integers and decimal numbers can be limited by providing a minimal and maximal value. These restrictions of the value range guarantee the quality and syntactic correctness of the information contained in FAIR-DOs, which benefits machine-interpretability.

To make *atomic data types* and their potential association with operations reusable and consistent, we introduce a simple hierarchical inheritance mechanism: They can optionally refer to at most one parent, which is intended to have a broader definition. Values must be correctly validated against all ancestors in the inheritance hierarchy.

2) Type Profiles: Type profiles specify the structure and content of a FAIR-DO by associating a set of typed *attributes* that are instances of the *Attribute* class. *Attributes* represent *data types* and additional semantics, which will be explained in more detail in Section III-B. The validation policy determines which combination of *attributes* must be in a FAIR-DO to comply with a *type profile*. The options are to expect none, exactly one, at least one, or all of the attributes. Furthermore, it is possible to allow or forbid additional *attributes* in the FAIR-DO that go beyond what is specified in the *type profile*. *Type Profiles* are used to describe the entire structure of FAIR-DO records. However, they can also specify the schema of complex JSON objects that are used as values of a specific *attribute* within a FAIR-DO record. This is useful when dealing with intricate or tightly coupled information that needs to be processed together but does not warrant a separate FAIR-DO (comparable to value objects). For example, describing a measurement requires storing a value together with a unit, possibly including some information about its accuracy.

Type profiles support multiple inheritance. This is known to cause problems, such as naming conflicts in programming languages [31]. For our FAIR-DO typing model, this does not apply because every *data type* and *attribute* is assigned a PID, making them unambiguously addressable. Remaining potential conflicts of multi-inheritance are solved through heuristics, whose implementation details are outside the scope of this work. *Type profiles* are therefore comparable to classes in OOP.

B. Attributes

An *attribute* points to a *data type* that defines its value range and may specify a default value. *Attributes* specify their cardinality by providing a lower boundary l and optionally an upper boundary u. This enables them to represent optional single values ($l = 0$; $u = 1$), mandatory single values ($l = 1$; $u = 1$), limited lists ($l \geq 0$; $u \geq 2$), and unlimited lists ($l \geq 0$; $u = \infty$) of values. *Attributes* behave covariantly when they are used in FAIR-DO information records or as a return value of an *operation* as detailed in Section III-C1. Since *attributes* are assigned a PID and inherit from the *Administrative Metadata* class, they can be referenced directly within a FAIR-DO record. Direct references to *attributes* prevent name conflicts when multiple *attributes* in a *type profile* share a *data type*. For instance, the Helmholtz KIP [9] includes "dateCreated" and "dateModified", both adhering to the ISO 8601 standard, which is represented as an *atomic data type*. Without direct references to *attributes*, both would resolve to the same PID (of the ISO 8601 *atomic data type*), losing semantic distinction and creating ambiguity.

This approach to *attributes* resembles object attributes or variables in OOP, both in terms of functionality and semantics. Our model uses *attributes* as the exclusive mechanism to associate FAIR-DO Operations with *data types*.

C. Modeling type-associated FAIR Digital Object Operations

FAIR-DO Operations must be agnostic to the technologies used by the environment in which they are executed (e.g., workflow engine, virtualization technology, operating system, networking). However, a variety of established languages, runtimes, tools, and other technologies must be usable within *operations*. Each technology may be installed, configured, and executed differently for each executing system. This depends only on the technology and the executing system, making it desirable to reuse this work in multiple *operations*.

To achieve this, we introduce *technology interfaces* that abstract the specific installation, configuration, and execution of each technology for different environments. This approach maintains the agnosticism of *operations* to their execution environment. As a result, the technologies used within *operations* become more maintainable, highly reusable, and system-specific optimizations can be applied. In Section III-C3, we describe technologies as *technology interfaces* with their environment-dependent implementations, which are represented by Adapter FAIR-DOs.

Figure 2 is a visual example of a possible application of a FAIR-DO Operation. The depicted operation, modeled using instances of the *Operation* and *Operation Step* classes, accepts an ORCiD via the "contact" attribute, which is present in the Helmholtz KIP [9]. Subsequently, it extracts the ORCiD number sequence and returns the "primary e-mail address" associated with the ORCiD profile as a result. For execution, two distinct technologies are used, each modeled by an instance of the *Technology Interface* class (as described in Section III-C3). These technologies are a regular expression (Regex) and a Python script.

1) Operations: The *Operation* class describes an action that can be performed on an *attribute*. Therefore, it must refer to the *attribute* it accepts and all *attributes* it returns. It may also refer to *attributes* that resemble environment variables such as API keys or configuration options. *Operations* always contain a non-empty ordered list of *operation steps* (described in Section III-C2), which specify all the tasks that are performed during the execution of an *operation*.

Compared to OOP concepts, *operations* are similar to both functions and methods with exactly one input parameter and possibly multiple return values. They are bound to *attributes* which they are executable on, thereby resembling methods. However, since they do not have a built-in capability to directly modify the associated values stored within the FAIR-DOs and are stateless, they align more closely with the typical characteristics of a function. Enabling concurrent execution of *operations* including *operation steps* and providing behavioral guarantees, especially with respect to ACID [32] compliance, is beyond the scope of this work.

2) Operation Steps: Operation steps can be understood as tasks within an *operation* workflow. The order in which these *operation steps* are executed in an *operation* is determined by their index and the availability of their *attributes*. Each *operation step* specifies whether it uses a *technology interface* (Section III-C3), calls another *operation*, or contains multiple *operation steps* itself. In addition, it includes a set of input and output *attribute mappings* (Section III-C4) to connect, transform, and specify values of *attributes*, depending on the *technology interface* or *operation* used in the *operation step*.

Due to their strong coupling with the scope of the *operations*, *operation steps* and *attribute mappings* are not reusable, are not assigned a PID, and are composites belonging to an *operation*. In contrast, *attributes*, *technology interfaces*, and *operations* heavily rely on their reusability and therefore are assigned a PID. *Operation steps* are comparable to function calls on a *technology interface* or another *operation*. However, they can also be viewed as subroutines when they contain *operation steps*.

3) Technology Interface: We separate technologies (e.g., languages, libraries, tools) from *operations* to enable reuse of technologies by multiple operations (n-to-m relationship). *Technology interfaces* provide an environment-independent interface to the execution of technologies. *Technology interfaces* specify a set of input *attributes*, a set of output *attributes*, and reference a set of Adapter FAIR-DOs via their PIDs.

Adapter FAIR-DOs contain environment-specific execution details and represent how the *technology interface* is executed on any given system. For example, the "Python" technology interface could have an Adapter FAIR-DO for a Docker-based executing system. This adapter would refer to a script that initiates a container using the inputs provided by the *technology interface* and processes the outputs generated by the container. We model these adapters as FAIR-DOs and not as classes in our model because the information needed for execution likely varies between executing systems, must be validated, persistently identified, and persistently stored As such, each

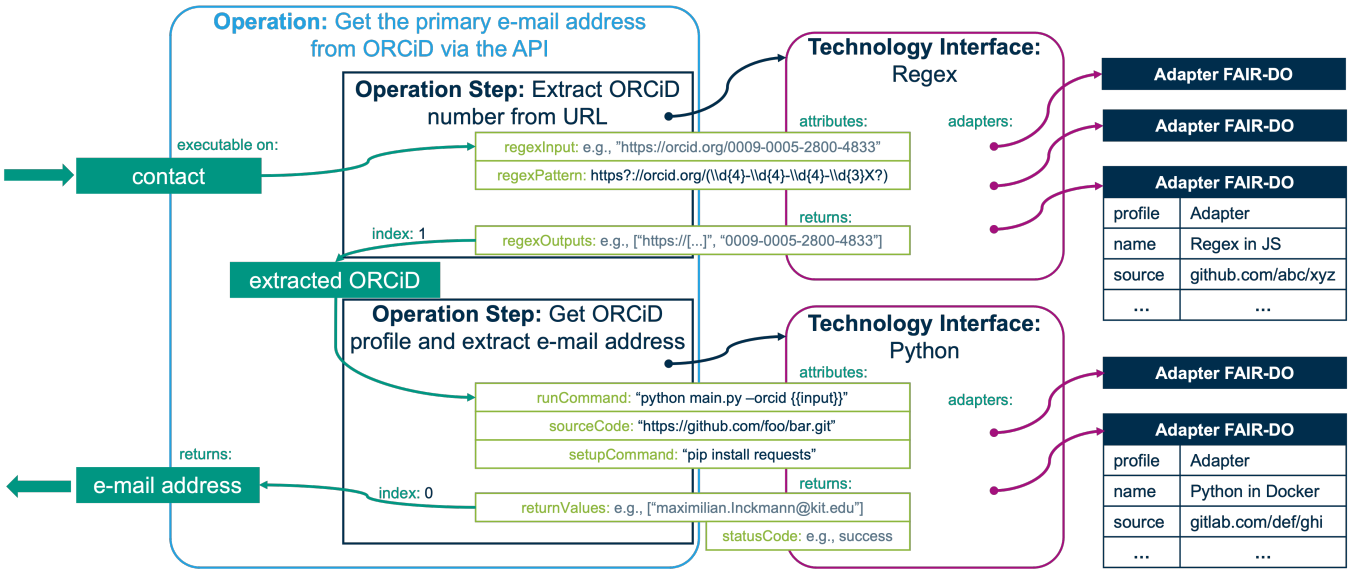

Fig. 2. Example of a FAIR-DO Operation and the connection of the entities described in Section III-C

system should specify a *type profile* to ensure that the Adapter FAIR-DOs contain all the information necessary for execution. This enables Adapter FAIR-DOs to be automatically resolved, verified in their integrity, and executed, being subject to the security policies of the executing system.

Applying this to the regular expressions used in our example (Figure 2), we observe that separate libraries and APIs exist for different environments. Particularly, we define a *technology interface* for "Regex" which accepts an input string and a pattern as parameters and returns an array of strings representing the regex groups. According to regex logic, the first element in the array will always indicate the fully validated string. For this "Regex" *technology interface*, we can then implement adapters for multiple environments (e.g., a JavaScript web browser environment, a Python environment, a Java environment). The executing system can then select the most suitable adapter among the available ones.

When comparing *technology interfaces* to OOP primitives, we think of them as functional interfaces, as they only provide exactly one `execute`-function that has a set of parameters and a set of return values. These functional interfaces may then be implemented by the adapters that can be injected into the executing service as a dependency, resembling the dependency inversion principle of OOP.

4) Attribute Mappings: Since the *attributes* provided as input to an *operation* are not necessarily identical to those specified by an *operation* or *technology interface*, we need to map between them. The *Attribute Mapping* class provides such a mapping mechanism and therefore enables the reuse of *operations* and *technology interfaces*. Figure 2 visualizes such a use case where an *operation* (light blue) requires a "contact" *attribute*, whose value is then transferred to the "regexInput" *attribute* of the "regex" *technology interface*. Similarly, an element from the output of the "regex" *technology interface* is extracted, transformed, and used in a subsequent *operation step*. We recall that every *attribute* conforms to a *data type* specifying its syntax. *Attribute mappings* therefore must fulfill multiple roles also known in OOP:

- **Defining constant values:** Not all *attributes* of *technology interfaces* need to be present in every FAIR-DO. Regex patterns, script locations, and setup details pertain to the operation itself, rather than the FAIR-DO it operates on. Therefore, *attribute mappings* support the specification of constant values within an *operation step* by providing a "value" field.
- **Type casting:** Different *attributes* often conform to different *data types*. For example, the "regexInput" *attribute* of the "Regex" *Technology Interface* naturally conforms to an arbitrary string. We therefore downcast the "contact" *attribute*, that conforms to the syntax of an ORCiD-URL, to a string to use it with the "Regex" *technology interface*. To make this mechanism work for both downcasting and upcasting, the executing system needs to validate the input values for the *attribute mappings* against the *data type* the output *attribute* complies to at runtime.
- **Addressing items in an array:** Figure 2 shows examples of n-to-1 upcasting. The "regexOutput" and "returnValues" *attributes* are arrays since their cardinality is greater than one. However, the "extracted ORCiD" and "e-mail address" *attributes* have a cardinality of exactly one, which requires the *attribute mappings* to select one element of their input arrays. The *attribute mappings* refer to the input and output *attribute* and specify the index of the element to be used. The validity has to be enforced by the executing system at runtime.

- **Providing templates for strings:** String templating, the process of adding the contents of a variable to a predefined string, is well known from programming and also of relevance in our operation model. In our example in Figure 2, we use this string template mechanism to insert the "extracted ORCiD" from the first *operation step* into the "runCommand" *attribute* that executes the Python script. The pattern of the insertion position (by default *"{{input}}"*) can be changed to facilitate as many use cases as possible.

IV. Model Implementation

We introduce IDORIS as a prototypical implementation of our typing model that can be found on GitHub[7]. The full name "Integrated Data Type and Operations Registry with Inheritance System" reflects the essential functionalities of our typing model described in Section III. Technologically, IDORIS is a Spring Boot[8] microservice, developed in Java 21[9]. For storage, the Neo4J graph database[10] is used in combination with Spring Data Neo4j[11]. IDORIS is designed as a Spring Modulith[12] that uses a traditional Model-Service-Controller architecture. Currently, we only provide a RESTful API that provides HATEOAS information for all endpoints using traditional Spring Web MVC controllers[13]. This API provides basic CRUD functionality and advanced features that require additional logic, such as resolving the inheritance hierarchy and retrieving available *operations* for *data types*.

A. Graph database

Due to the high interconnectivity of our model, efficient querying of relationships between model components is essential. This is a typical use case for graph databases. We chose Neo4j for its labeled-property graph model and integrability into the Spring framework. This enables us to easily model an expressive graph that closely resembles our typing model with only minor technical changes. IDORIS uses efficient in-database processing capabilities to find all *operations* executable on an *attribute* or transitively a *data type*, to detect cycles in the graph, and to resolve inheritance hierarchies. Furthermore, we can use graph algorithms for path finding, cycle detection, and relationship querying provided by Neo4j directly inside our graph database.

B. Rule-based validation and processing

Since our model is highly dependent on the accuracy of user-provided information, IDORIS must validate this data both syntactically and semantically. A capable validation mechanism is essential for implementing inheritance mechanisms for atomic data types and type profiles. It should validate not only individual entities, but also their contextual relationships. This is clearly beyond the capabilities of a JSON schema.

We therefore implemented a rule-based and, therefore, highly modular approach for validation to enhance maintainability. This is accomplished by using the "Visitor" design pattern [33], which separates logic from model classes in a highly modular manner. Consequently, it is used, among others, for semantic validation and optimization within compilers. Each validation rule is implemented in a separate Visitor class, having a dedicated behavior for each model class it is called upon (e.g., *Atomic Data Type*, *Type Profile*, *Operation*). Visitors perform non-trivial validations of the inheritance hierarchy and of relations to other entities (such as *attributes*). This is primarily done through recursion, interaction with the accessor methods, and interaction with the graph database to ensure cross-entity consistency. This approach ensures that the inheritance hierarchy is free from conflicts and circular dependencies. In IDORIS, Visitors are currently only used for validation purposes, but are designed to solve future problems such as JSON schema generation or optimization algorithms.

V. Evaluation and Discussion

A. IDORIS-based Use Case

We visualized our typing model in Figure 1. In Section III-C, we defined our approach for type-associated FAIR-DO Operations. Figure 2 illustrates a running example. In this subsection, we will reuse the running example as a use case. We will also provide a corresponding excerpt of the actual graph data structure in Figure 3 (as described in Section IV-A). This graph represents an operation with all relevant nodes and relations. For clarity, we omit the node properties and instead show a description of the contents of each node. The color labels of the nodes in our labeled-property graph align with those of Figures 1 and 2. Specifically, the *operation* and its *operation steps* are represented in light blue, *attributes* in dark green, *attribute mappings* in light green, and *technology interfaces* in purple. This graph can be retrieved by executing the simple Cypher-Query: `MATCH (n: Operation | AttributeMapping | Operation Step | TechnologyInterface | Attribute) RETURN n)`. Therefore, we can argue that the graph directly represents the classes of our UML-based typing model in a semantically meaningful manner.

The semantically meaningful nodes and relations in our graph database can be leveraged to perform more complex queries. For example, in Figure 3, the red relations form a cycle within the graph representing the flow of data within the *"Get primary e-mail from ORCiD via API" operation*. It is evident how the data traverses from the *"contact" attribute* through the *attribute mapping* into the *"regexInput" attribute*, which is then processed by the *"Regex" technology interface*. The output of this *technology interface* is subsequently transformed and inserted into a command that initiates a "Python script" that outputs the "e-mail address". This cycle is queried using the Cypher query:

[7]https://github.com/kit-data-manager/idoris
[8]https://spring.io
[9]https://java.com
[10]https://neo4j.com
[11]https://docs.spring.io/spring-data/neo4j/reference
[12]https://docs.spring.io/spring-modulith/reference
[13]https://docs.spring.io/spring-framework/reference/web/webmvc.html

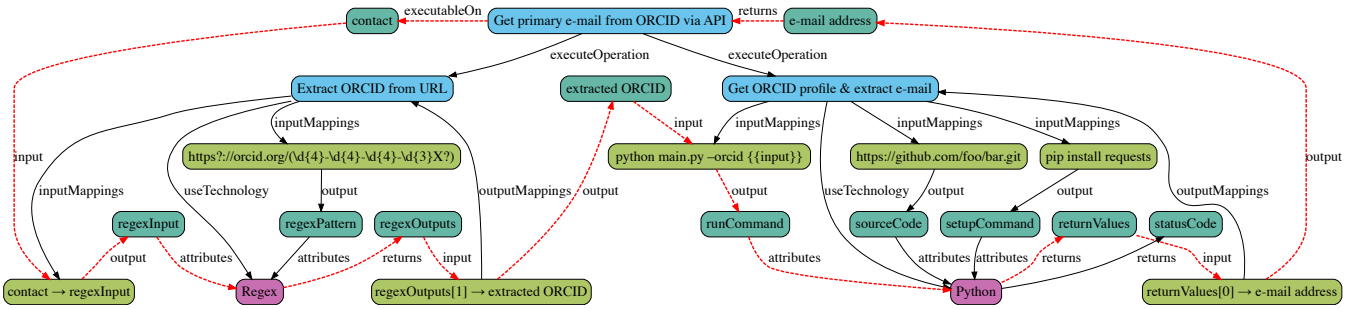

Fig. 3. Excerpt from the Graph representing a FAIR-DO Operation stored in the database

`MATCH (m1) WITH collect(m1) as nodes CALL apoc.nodes.cycles(nodes) YIELD path RETURN path)`. The same query returns an additional cycle for each *operation step*, representing levels of abstraction within an *operation*. This cycle detection is useful for a future executing system to automatically parallelize processing and to ensure data is available when needed. The concrete use case of this example can be simplified by using a platform-independent execution mechanism, such as Web Assembly (WASM) [34], whose limitations are outside the scope of this work. However, more complex use cases that need mechanisms for environment-specific execution, optimization, and access to resources that only native code can provide may use technologies such as Docker Containers and Python Scripts, demanding a more flexible modeling approach. Our intention is not to develop a "universal programming language", but rather to facilitate the diverse range of languages, frameworks, and tools that already exist through our technology-agnostic model for FAIR-DO Operations.

IDORIS must also ensure that no invalid cycles are introduced. Examples of such unwanted circular dependencies include cycles in the inheritance hierarchy, *type profiles* that use themselves as *attributes*, and *operations* calling themselves within *operation steps*. To avoid this, we used Neo4j pathfinding algorithms and created a rule for IDORIS' rule-based validator system (Section IV-B). In Listing 1, we show an excerpt of such a validator that also creates error messages through the API including severity level, message, and entity of interest. This feature of IDORIS provides users with detailed error messages when creating new elements and ensures data integrity. By implementing separate validator classes for each rule, we improve the maintainability of our codebase through a separation of concerns. These validators can also be used to ensure semantic correctness, for example, by ensuring conflict-freeness in the inheritance hierarchies of *atomic data types* and *type profiles*.

Listing 1. Excerpt of the Acyclicity Validator role in Java syntax

```
@Rule(
  appliesTo = {AtomicDataType.class, TypeProfile.
      class},
  tasks = {RuleTask.VALIDATE},
  dependsOn = {SyntaxValidator.class}
)
public class AcyclicityValidator extends
    ValidationVisitor {
  @Autowired // Dependency Injection
  private Neo4jClient neo4jClient;

  @Override // Behavior for Atomic Data Types
  public ValidationResult visit(AtomicDataType
      atomicDataType, Object... args) {
    return doesNotInheritItself(atomicDataType);
  }

  @Override // Behavior for Type Profiles
  public ValidationResult visit(TypeProfile profile,
       Object... args) {
    return ValidationResult.combine(
      doesNotInheritItself(profile),
      doesNotUseItselfAsAttribute(profile));
  }

  private ValidationResult doesNotInheritItself(
      DataType dataType) {
    String query = "MATCH path = (n:DataType {id:
        $nodeID})-[:inheritsFrom*1..]->(n) RETURN
        path";
    // Query the path from the Neo4j database
    var path = neo4jClient.query(query)
      .bind(dataType.getId()).to("nodeID")
      .fetch().all();
    if (!path.isEmpty()) { // Return error with path
      return ValidationResult.error("Circular
          inheritance detected", path);
    } else { // Return success message
      return ValidationResult.ok();
    }
  }[...]
}
```

Figure 4 shows detailed examples of two application cases of *Type Profiles* (described in Section III-A2 and marked in red): The "Helmholtz Kernel Information Profile" *type profile* is illustrated with excerpts containing selected *attributes*. This profile is used to describe the content of the FAIR-DOs that adhere to it. One of these *attributes* (Section III-B) conforms to the "Checksum" *type profile*. This profile is used to describe a complex JSON object embedded within a value in a FAIR-DO, utilizing the "Helmholtz Kernel Information Profile". The resulting JSON object contains the hash and the algorithm that generated the hash. Furthermore, it shows an example for the inheritance of *atomic data types* (Section III-A1) by specifying that "ORCiD-URL" inherits from "URL".

Furthermore, Figure 4 visualizes *data types*, namely *atomic data types* (yellow) and *type profiles* (red), in addition to the

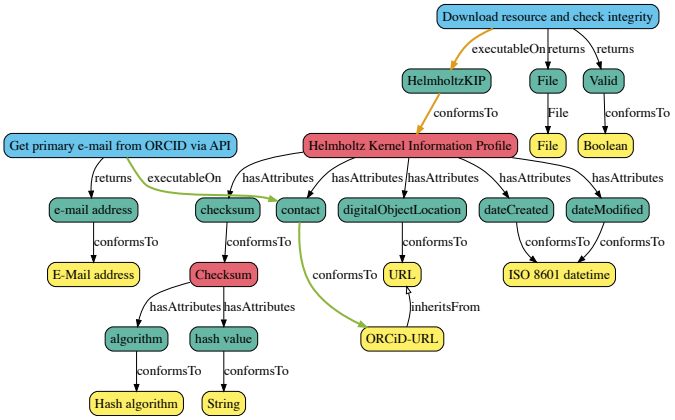

Fig. 4. Excerpt from the Graph representing Data Types in the graph database

attributes, and *operations* using the running example and an additional example operation "Download Resource and Check Integrity". We use these to elaborate on the fulfillment of conceptual association mechanisms for operations and FAIR-DOs according to "Record typing", "Profile typing" and "Attribute typing" (as introduced in [29]). Since our *operations* are assigned with PIDs, we support referencing them from FAIR-DOs, enabling "Record typing". We realize both "Profile typing" and "Attribute typing" by specifying a single *attribute* an *operation* is executable on. For "Profile typing", this *attribute* conforms to a *type profile* (orange path in Figure 4). For "Attribute typing", this *attribute* can conform to any *data type*, namely *atomic data types* or a *type profile* (light green path in Figure 4). However, we opted not to adopt the duck typing variant of "Attribute Typing" by permitting exactly one *attribute* to be executed by an *operation*. Hence, all FAIR-DO operation association mechanisms are (at least partially) supported by our model and implemented in IDORIS.

B. Comparison to Existing Data Type Registry Models

Our typing model was designed based on concepts from the ePIC and EOSC DTRs [11]. The core concepts — defining the syntax of simple values and structuring complex values — remain unchanged. In the ePIC and EOSC DTRs, *PID-BasicInfoTypes* and *PID-InfoTypes* are sometimes called Data Types for simplicity. *PID-BasicInfoTypes* for the syntax of simple values are modeled by the *Atomic Data Type* class (Section III-A1). To define complex structures, our model combines *PID-InfoTypes* (for complex JSON values inside a FAIR-DO) and *KernelInformationProfiles* (for the structure of FAIR-DOs) into the *Type Profile* class (Section III-A2). As a new approach, the *Data Type* class abstracts *Atomic Data Types* and *Type Profiles*. This reduces redundancy and clarifies the semantics of "data types". The newly introduced inheritance mechanisms for both *Atomic Data Types* and *Type Profiles* promote the reusability of their instances. This already enables basic polymorphic behavior through subtyping, facilitating reuse of the association between *data types* and *operations*. This basic polymorphic behavior relates to the model's ability to specify machine-actionable *operations* that are associated with *attributes* (and transitively *data types*), enabling type-associated FAIR-DO Operations. *Atomic Data Types* omit special fields from *PID-BasicInfoTypes* that specify measurement units or categories, as well as other rarely used or undocumented fields, to improve the semantic clarity of our typing model. Similarly, the former *SubSchemaRelation* for PITs and KIPs [11] was split into a validation policy and an "allow additional attributes" flag.

Unlike the ePIC and EOSC DTRs, IDORIS is not based on JSON schemas. Instead, we use a graph database, ideal for storing highly connected entities and executing graph algorithms to query the inheritance hierarchy of data types and for finding *operations* executable on an *attribute* or *data type*. We also provide a more capable rule-based validation logic that is able to validate more than just the syntax of single entities. This enables us to manage type-associated operations and ensure the quality of the information stored in IDORIS. We decided against an RDF-based system due to its limited integration into the Spring framework and its higher modeling complexity. Although RDF-based graph databases, such as Apache Jena[14], offer potential advantages, including easier integration into knowledge graphs, the ability to reuse terms from ontologies, and possible support for federated queries beyond single systems, concrete use cases benefiting from these features have not yet been identified.

VI. Conclusion and Future Work

Our typing model contributes to the long-term vision of machine-actionable FAIR-DOs by introducing type-associated FAIR-DO Operations that are well-described in a technology-agnostic and highly reusable manner. We improved on the models used in existing DTRs by integrating inheritance mechanisms, refactoring the components of the type system, and providing mechanisms that associate data types with FAIR-DO Operations. To demonstrate the feasibility and capabilities of our model, we developed IDORIS, a prototype for a next-generation of DTRs that can accommodate data types, technology-agnostic FAIR-DO Operations, and perform computations to associate them. In addition, IDORIS incorporates a robust validation system that provides detailed feedback to ensure data integrity. Our model and IDORIS therefore provide a foundational typing infrastructure, improving interoperability, reusability, and automation in research data management, and enabling future work towards executing FAIR-DO Operations.

Upcoming research includes developing execution components for FAIR-DO Operations, including adapters for widely adopted technologies, enhancing system scalability, and integrating federated DTR instances. We will contribute these results to international and national initiatives such as the Research Data Alliance (RDA), the FAIR-DO Forum, the German National Research Data Infrastructure (NFDI), and the Helmholtz Metadata Collaboration Platform (HMC).

[14]https://jena.apache.org

References

[1] European Commission: Directorate-General for Research and Innovation, S. Hodson, S. Jones, P. Wittenburg, S. Collins, F. Genova, N. Harrower, L. Laaksonen, and R. Petrauskaité, "Turning FAIR into reality: Final Report and Action Plan from the European Commission Expert Group on FAIR Data," European Commission Directorate General for Research and Innovation Directorate B – Open Innovation and Open Science Unit B2 – Open Science, Brussels, Tech. Rep., 2018. [Online]. Available: https://data.europa.eu/doi/10.2777/1524

[2] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, J. Bouwman, A. J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C. T. Evelo, R. Finkers, A. Gonzalez-Beltran, A. J. G. Gray, P. Groth, C. Goble, J. S. Grethe, J. Heringa, P. A. C. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S. J. Lusher, M. E. Martone, A. Mons, A. L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M. A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons, "The FAIR Guiding Principles for scientific data management and stewardship," *Scientific Data*, vol. 3, no. 1, p. 160018, Mar. 2016. [Online]. Available: https://www.nature.com/articles/sdata201618

[3] I. Anders, C. Blanchi, B. Daan, M. Hellström, S. Islam, T. Jejkal, L. Lannom, K. Peters-von Gehlen, R. Quick, A. Schlemmer, U. Schwardmann, S. Soiland-Reyes, G. Strawn, D. van Uytvanck, C. Weiland, P. Wittenburg, and C. Zwölf, "FAIR digital object technical overview," FAIR Digital Objects Forum, Proposed Endorsed Note, Apr. 2023. [Online]. Available: https://zenodo.org/records/7824714

[4] E. Schultes and P. Wittenburg, "FAIR Principles and Digital Objects: Accelerating Convergence on a Data Infrastructure," in *Data Analytics and Management in Data Intensive Domains*, ser. Communications in Computer and Information Science, Y. Manolopoulos and S. Stupnikov, Eds. Cham: Springer International Publishing, 2019, pp. 3–16.

[5] N. Blumenröhr, P.-J. Ost, F. Kraus, and A. Streit, "FAIR Digital Objects for the Realization of Globally Aligned Data Spaces," in *2024 IEEE International Conference on Big Data*. Washington, DC, USA: IEEE, Dec. 2024, pp. 374–383. [Online]. Available: https://ieeexplore.ieee.org/document/10825796

[6] S. Sun, L. Lannom, and B. P. Boesch, "Handle system overview," Internet Engineering Task Force, Informational RFC 3650, Nov. 2003. [Online]. Available: https://datatracker.ietf.org/doc/rfc3650/

[7] S. Sun, S. Reilly, and L. Lannom, "Handle system namespace and service definition," Internet Engineering Task Force, Informational RFC 3651, Nov. 2003. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3651

[8] S. Sun, S. Reilly, L. Lannom, and J. Petrone, "Handle system protocol (ver 2.1) specification," Internet Engineering Task Force, Informational RFC 3652, Nov. 2003. [Online]. Available: https://datatracker.ietf.org/doc/rfc3652/

[9] C. Curdt, G. Günther, T. Jejkal, C. Koch, F. Krebs, A. Pfeil, A. Pirogov, J. Schweikert, P. Videgain Barranco, and M. Weinelt, "Helmholtz Metadata Collaboration, Helmholtz Kernel Information Profile," HMC Office, GEOMAR Helmholtz Centre for Ocean Research, Kiel, Germany, Report, Dec. 2022. [Online]. Available: https://oceanrep.geomar.de/id/eprint/57942/

[10] C. Weiland, S. Islam, D. Broder, I. Anders, and P. Wittenburg, "FDO Machine Actionability," Nov. 2022. [Online]. Available: https://zenodo.org/records/7825650

[11] U. Schwardmann, "Automated schema extraction for PID information types," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec. 2016, pp. 3036–3044. [Online]. Available: https://ieeexplore.ieee.org/document/7840957

[12] C. Bizer, T. Heath, and T. Berners-Lee, "Linked Data - The Story So Far:," *International Journal on Semantic Web and Information Systems*, vol. 5, no. 3, pp. 1–22, Jul. 2009. [Online]. Available: https://doi.org/10.4018/jswis.2009081901

[13] R. Ravinder and L. J. Castro, "An introduction to (webby) FAIR Digital Objects - A machine-actionable approach to FAIR," Feb. 2025. [Online]. Available: https://zenodo.org/records/14882665

[14] P. Groth, A. Gibson, and J. Velterop, "The anatomy of a nanopublication," *Information Services and Use*, vol. 30, no. 1-2, pp. 51–56, Feb. 2010. [Online]. Available: https://doi.org/10.3233/ISU-2010-0613

[15] K. De Smedt, D. Koureas, and P. Wittenburg, "FAIR Digital Objects for Science: From Data Pieces to Actionable Knowledge Units," *Publications*, vol. 8, no. 2, p. 21, Apr. 2020. [Online]. Available: https://www.mdpi.com/2304-6775/8/2/21

[16] P. Lemieux, "Entertainment Identifier Registry (EIDR) URN Namespace Definition," Internet Engineering Task Force, Informational RFC7972, Sep. 2016. [Online]. Available: https://datatracker.ietf.org/doc/rfc7972

[17] E. I. R. EIDR, "Introduction to the EIDR Data Model," Jul. 2020. [Online]. Available: https://www.eidr.org/documents/Introduction%20to%20the%20EIDR%20Data%20Model.pdf

[18] W3C OWL Working Group, "OWL 2 Web Ontology Language Document Overview (Second Edition)," World Wide Web Consortium (W3C), Recommendation, Dec. 2012. [Online]. Available: https://www.w3.org/TR/owl2-overview/

[19] H. Knublauch and D. Kontokostas, "Shapes Constraint Language (SHACL)," World Wide Web Consortium (W3C), Recommendation, Jul. 2017. [Online]. Available: https://www.w3.org/TR/shacl/

[20] T. Weigel, B. Plale, M. Parsons, G. Zhou, Y. Luo, U. Schwardmann, R. Quick, M. Hellström, and K. Kurakawa, "RDA Recommendation on PID Kernel Information," Research Data Alliance, Recommendation, Nov. 2019. [Online]. Available: https://zenodo.org/records/3581275

[21] T. Weigel, T. DiLauro, and T. Zastrow, "PID Information Types WG final deliverable," Research Data Alliance, Working Group Output, Jul. 2015. [Online]. Available: https://doi.org/10.15497/FDAA09D5-5ED0-403D-B97A-2675E1EBE786

[22] L. Lannom, D. Broeder, and G. Manepalli, "RDA Data Type Registries Working Group Output," Research Data Alliance, Working Group Output, Apr. 2015. [Online]. Available: https://zenodo.org/records/1406127

[23] European Commission, "Commission Implementing Decision (EU) 2017/1358 of 20 July 2017 on the identification of ICT Technical Specifications for referencing in public procurement," pp. 16–19, Jul. 2017. [Online]. Available: https://eur-lex.europa.eu/eli/dec_impl/2017/1358/oj

[24] L. Ávila Calderón, Y. Shakeel, A. Gedsun, M. Forti, S. Hunke, Y. Han, T. Hammerschmidt, R. Aversa, J. Olbricht, M. Chmielowski, R. Stotzka, E. Bitzek, T. Hickel, and B. Skrotzki, "Management of reference data in materials science and engineering exemplified for creep data of a single-crystalline Ni-based superalloy," *Acta Materialia*, vol. 286, p. 120735, Mar. 2025. [Online]. Available: https://doi.org/10.1016/j.actamat.2025.120735

[25] F. Kraus, N. Blumenröhr, G. Götzelmann, D. Tonne, and A. Streit, "A Gold Standard Benchmark Dataset for Digital Humanities," in *OM-2024: The 19th International Workshop on Ontology Matching Collocated with the 23rd International Semantic Web Conference (ISWC 2024), November 11th, Baltimore, USA*, ser. CEUR Workshop Proceedings, vol. 3897. Baltimore, MD, USA: RWTH Aachen, Nov. 2024, pp. 1–17. [Online]. Available: https://doi.org/10.5445/IR/1000178023

[26] Z. Mayer, J. Kahn, M. Götz, Y. Hou, T. Beiersdörfer, N. Blumenröhr, R. Volk, A. Streit, and F. Schultmann, "Thermal Bridges on Building Rooftops," *Scientific Data*, vol. 10, no. 1, p. 268, May 2023. [Online]. Available: https://www.nature.com/articles/s41597-023-02140-z

[27] N. Blumenröhr and F. Kraus, "Leveraging Large Language Models for Processing and Evaluating FAIR Digital Objects," in *Joint Proceedings of the ESWC 2025 Workshops and Tutorials Co-Located with 22nd Extended Semantic Web Conference (ESWC 2025)*, ser. CEUR Workshop Proceedings, vol. 3977. Portorož, Slovenia: CEUR-WS.org, Jun. 2025, p. 7. [Online]. Available: https://ceur-ws.org/Vol-3977/NSLP-10.pdf

[28] R. Tupelo-Schneck, "An Introduction to Cordra," *Research Ideas and Outcomes*, vol. 8, p. e95966, Oct. 2022. [Online]. Available: https://riojournal.com/article/95966/

[29] N. Blumenröhr, J. Böhm, P. Ost, M. Kulüke, P. Wittenburg, C. Blanchi, S. Bingert, and U. Schwardmann, "A Comparative Analysis of Modeling Approaches for the Association of FAIR Digital Objects Operations," Apr. 2025. [Online]. Available: https://arxiv.org/abs/2504.05361

[30] R. E. Kahn, C. Blanchi, L. Lannom, P. A. Lyons, G. Manepalli, R. Tupelo-Schneck, and S. Sun, "Digital Object Interface Protocol (DOIP) Specification version 2.0," DONA Foundation, Specifications, Nov. 2018. [Online]. Available: https://www.dona.net/sites/default/files/2018-11/DOIPv2Spec_1.pdf

[31] R. C. Martin, "Java and C++ - A critical comparison," Mar. 1997.

[Online]. Available: https://web.archive.org/web/20051024230813/http://www.objectmentor.com/resources/articles/javacpp.pdf
[32] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, no. 4, pp. 287–317, Dec. 1983. [Online]. Available: https://dl.acm.org/doi/10.1145/289.291
[33] E. Gamma, R. Helm, R. Johnston, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, 1995.
[34] WebAssembly Working Group, "WebAssembly Core Specification," World Wide Web Consortium (W3C), W3C Recommendation REC-wasm-core-1-20191205, Dec. 2019. [Online]. Available: https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/