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. To be machine-actionable, they must support operations that interact with their contents. This can be achieved by associating operations with FAIR-DO data types. However, current typing models and Data Type Registries lack support for type-associated operations. In this work, we introduce a typing model that describes type-associated and technology-agnostic FAIR Digital Object Operations in a machine-actionable way, building and improving on the existing concepts. In addition, we introduce the Integrated Data Type and Operations Registry with Inheritance System, a prototypical implementation of this model that integrates inheritance mechanisms for data types, a rule-based validation system, and the computation of type-operation associations. Our approach significantly improves the machine-actionability of FAIR Digital Objects, paving the way towards dynamic, interoperable, and reproducible research workflows.**

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

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]. To address these challenges, the FAIR principles [2] were established, guiding best practices for sustainable and efficient management of research data.

FAIR Digital Objects (FAIR-DOs) [3], [4] embody these principles, aiming to provide common mechanisms to enable machine-actionable, persistent, and harmonized representation of (meta)data beyond the borders of data spaces [1], [5]. FAIR-DOs use globally unique, resolvable, and persistent identifiers (PIDs) with their persistent records based on the well-established Handle system [6], which ensures their longevity and reliable referencing. Every value inside a FAIR-DO record is assigned a data type that is always referenced using a PID and defines the syntax as well as the semantic meaning of this value. This data type should be reused wherever its syntax and semantics fit, ensuring that identical references denote identical meaning. This generates harmonized artifacts which are interpretable and processable by a machine to, e.g., determine available operations for FAIR-DOs. Multiple data types may be aggregated in profiles that define the structure of a FAIR-DO record. Beyond the borders of research domains, domain-agnostic profiles, such as the Helmholtz Kernel Information Profile [7], are used to harmonize essential information in FAIR-DOs. The 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], [8].

FAIR-DO Operations describe a mechanism for type-based interaction with FAIR-DOs and their contents, thus making them machine-actionable. To allow for automatic execution, they need to be described in a fully typed, interoperable, technology-agnostic, and reusable manner. To enable the computation of available FAIR-DO Operations for a given FAIR-DO, we need to bidirectionally associate them with data types in a type system. This leads to a highly inter-connected typing model that needs to be managed, queried, and validated.

Existing Data Type Registries (DTRs) [9] with their typing models represent a significant development towards machine-interpretable FAIR-DOs. However, their schema-reliant architecture makes them unable to utilize and provide complex mechanisms beyond the capabilities of JSON schema, and thus cannot facilitate type-associated FAIR-DO Operations. Such capabilities are needed to, e.g., bidirectionally associate FAIR-DO Operations to data types, to realize inheritance mechanisms and to deal with the highly connected typing model that is required for FAIR-DO Operations.

To address these shortcomings, we developed a typing model for a new graph-based FAIR-DO type system that we prototypically implemented as the Integrated Data Type and Operations Registry with Inheritance System (IDORIS) to showcase its feasibility. This typing model is conceptually based on the components of current DTR systems and lessons learned through their usage. We leverage the resulting type system to model type-associated FAIR-DO Operations in a technology-agnostic, highly reusable, and well-described

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.

manner. We come up with a comprehensive solution that integrates FAIR-DO Operations as type-associated operations, inheritance, and semantic validation within a single type system. Hence, this work contributes to the field of FAIR (research) data management by achieving substantial progress in the long-term vision of machine-actionability.

The paper is organized as follows: Section II provides a review of relevant technologies and related work. In Section III, we describe our typing model as a basis for type-associated FAIR-DO Operations and in Section IV we introduce IDORIS as a prototypical implementation of this model. In Section V, we discuss and evaluate our model as well as the prototypical implementation based on a specific use-case. Finally, Section VI summarizes the key contributions and discusses future research directions.

II. Related Work

FAIR-DOs essentially constitute a data management approach comparable to Linked Data [10] or nanopublications [11], but are distinguished primarily by their emphasis on persistence and strong type-safety, which ensure their machine-interpretability. Within the Research Data Alliance (RDA)[1], multiple working groups and interest groups established outcomes and recommendations on FAIR-DO content [12], information typing models [13], and DTRs [14], which have been acknowledged, among others, by the European Commission [1], [15]. Despite ongoing discussions and early implementations within international initiatives such as the RDA and the FAIR Digital Objects Forum[2], comprehensive practical solutions addressing critical gaps in the FAIR-DO typing infrastructure still need to be developed and widely adopted. The existing typing infrastructure comprises three Data Type Registry instances – the ePIC test DTR[3], the ePIC production DTR[4] and the EOSC DTR[5]. Those DTRs follow the same typing model, with three schemas that constitute the basis for information typing: *PID-BasicInfoTypes*, *PID-InfoTypes* and *KernelInformationProfiles* [9], [12]; However, their implementations slightly differ between the DTR instances and are not standardized. These systems were already used to model information types for FAIR-DOs in the frame of several use cases from different domains, e.g., in material sciences [16], in digital humanities [17], and in energy research [18].

Technologically, all current DTRs are based on Cordra [19], a JSON schema-based metadata repository that is only able to validate the syntactic compliance to JSON schemas. Thus, the primary focus of existing DTRs has been limited to syntactic validation, offering at most rudimentary support for semantic validation or linking executable operations to specific data types. Moreover, despite the recognized benefits of object-oriented programming (OOP) principles such as inheritance and polymorphism in software engineering, current implementations of information types by DTRs either completely lack or inadequately support these mechanisms. This absence 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, the DTRs do not support systematic reuse of type definitions, significantly hindering scalability and maintainability. To enhance the domain-specific expressivity of FAIR-DOs, it can be desired to create a Kernel Information Profile (KIP) [12] of selected domain-specific attributes in addition to those provided by the domain-agnostic ones, e.g. Helmholtz KIP [7]. This case is not adequately supported by the existing DTRs, forcing creators of such domain-specific profiles to remodel the specification of the domain-agnostic KIP. This leads to redundant work, reduced reusability, and is a missed opportunity to leverage the de-facto subtyping relation between domain-specific and -agnostic KIPs.

Conceptually, FAIR-DO Operations provide a mechanism to interact with FAIR-DOs, i.e., the values contained within the Handle records, and the external resources they reference (i.e., the bit sequence) [8]. Currently, multiple approaches exist for service-oriented FAIR-DO Operations. They typically focus on basic CRUD (Create, Read, Update, Delete) functionalities as specified by the Digital Object Interface Protocol (DOIP) [20]. They operate at the level of the FAIR-DO as a whole, and must be individually implemented by each service that supports such operations [20]. Currently, there is no method to describe technology-agnostic FAIR-DO Operations independently from the specific executing service and to dynamically associate them to FAIR-DOs according to at least one FAIR-DO association mechanism, i.e., "Record typing", "Profile typing", and "Attribute typing", as described in [21].

These existing limitations highlight the necessity of 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

Before going into the specifics of our model, we need to provide a brief overview of the current technical implementation of typing for FAIR-DOs.

Every FAIR-DO is described by an information record of key-value pairs, stored in the Handle Registry[6] and resolvable by a Handle PID [5], [6]. 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. We use the term "typing" to refer to the availability of information within FAIR-DO records and our typing model. This is similar to the "information typing"

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

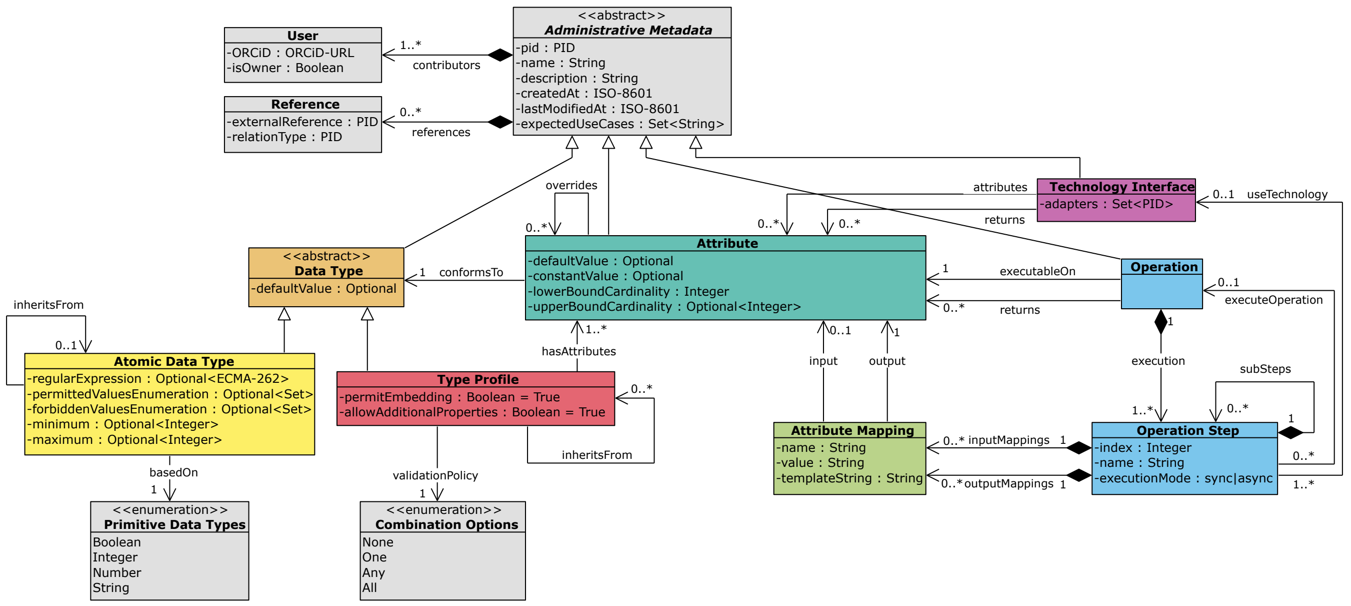

Fig. 1. UML model class diagram of the complete typing model

used within current DTRs, but opposite to "FAIR-DO typing" in the context of association mechanisms for operations as proposed in [21].

On this basis, we introduce our typing model for FAIR-DOs, the description of technology-agnostic FAIR-DO Operations and the association between data types and operations, indicating the analogies to OOP principles. Figure 1 depicts the typing model for FAIR-DOs in a colorized UML diagram, consisting of the following classes: *Data Type* as a generalized term (orange), *Atomic Data Type* (yellow), *Type Profile* (red), and *Attribute* (dark green). Likewise, technology-agnostic 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). The gray elements are enumerations and administrative metadata that partially depend on the implementation. For simplicity, we write the names of the classes in lowercase italics to refer to their instances, and in uppercase italics to refer to the classes themselves.

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* consistently, thereby reducing the redundancy and complexity of our model and its implementation whilst enhancing its semantic clarity and expressivity. Likewise, as detailed in Subsection III-B, this also allows the definition and logic of attributes in the *Attribute* class to be agnostic towards the instances of the *Data Type* subclasses they conform to.

1) Atomic Data Types: Instances of the *Atomic Data Type* class define the syntax of every value in the information record of any FAIR-DO. They are built on top of primitive JSON types (Boolean, Integer, Number, or String) to enable JSON serialization. Therefore, *atomic data types* are comparable to primitive data types in OOP, but offer additional restriction mechanisms that allow for a more strict validation of values: for any *atomic data type*, predefined constant enumerations of permitted and forbidden values can be specified, which are prioritized 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 space 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. Upon validation of a value for an *atomic data type*, this value needs to be correctly validated against all *atomic data types* in the inheritance chain.

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 further explained in Subsection III-B. The validation policy determines which combination of *attributes* must be available, and whether to allow or forbid additional *attributes* in a FAIR-DO complying with the *type profile*. The options are to allow none, exactly one, any but at least one, or all of the attributes. A *type profile* can describe the entire structure of FAIR-DO records and complex JSON objects that are used as values of a specific *attribute* within a FAIR-DO record. The latter option is particularly useful when dealing with intricate or tightly-coupled information that is not generic enough to be extracted into a separate FAIR-DO but still needs to be processed together. For instance, the description of measurement units requires storing a value, a unit, and possibly some information about its accuracy together.

In addition, instances of the *Type Profile* class can make use of a multi-inheritance mechanism. Despite being known to cause problems such as naming conflicts in programming languages [22], this is not the case in our model since every *data type* and *attribute* is assigned a PID, making them unambiguously addressable. The remaining potential conflicts of multi-inheritance can be 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 space and a default value, if any. *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 ($l \geq 0; u = \perp$) 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 III-C1. Since *attributes* are assigned a PID and contain elements of the *Administrative Metadata* class, they can be referenced directly to specify a value within a FAIR-DO record. This is necessary in case multiple values that conform to the same *atomic data type* are used in a *type profile*. For instance, the Helmholtz KIP [7] includes "dateCreated" and "dateModified", both adhering to the ISO 8601 standard, which is represented as an *atomic data type*. Without directly referencing these *attributes*, both values would refer to the identical PID of the ISO 8601 *atomic data type*, resulting in a loss of valuable semantic differentiation.

This approach to *attributes* resembles object attributes or variables in OOP, both in terms of functionality and semantics. However, *attributes* according to the *Attribute* class in our model additionally fulfill the crucial role of associating FAIR-DO Operations with *data types*.

C. Modeling FAIR Digital Object Operations

In the following, we outline the methodology for modeling technology-agnostic FAIR-DO Operations based on the previously introduced classes of the typing model for FAIR-DOs. For this, we need to abstract and describe these operations as well as technologies, enrich them with meaningful metadata, and provide a mechanism to adapt this generic definition to actual execution environments in order to enable automatic computation.

Figure 2 is a visual example of a possible application of a FAIR-DO Operation. The depicted operation, modeled by instances of the *Operation* and *Operation Step* classes, receives an ORCiD via the "contact" *attribute*, contained within the Helmholtz KIP [7], extracts the ORCiD number/letter sequence, and returns the "primary e-mail address" of the ORCiD profile as the result. During the execution, two distinct technologies are used that are modeled by instances of the *Technology Interface* class (described in Subsection III-C3): a regular expression (Regex) and a Python Script.

1) Operations: The *Operation* class describes an action that can be performed on an instance of the *Attribute* class to which it is applicable. Therefore, it must reference this *attribute* and all *attributes* it returns. *Operations* always contain a non-empty ordered list of instances of the *Operation Steps* class (described in Subsection III-C2), that specify all the tasks that are performed during the execution of an *operation*. When comparing FAIR-DO Operations to OOP, instances of the *Operation* class are similar to both, functions and methods with exactly one input parameter and possibly multiple return values. They are bound to instances of the *Attribute* class (or the respective *data type*) on which they are executable, thereby resembling methods. However, as they do not have the 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.

2) Operation Steps: Operation steps can be understood as tasks in an operation workflow. The order of execution of the *operation steps* within an *operation* is specified in ascending order by the index and the availability of the *attributes*. An *operation step* specifies whether it uses a technology via an instance of the *Technology Interface* class (Subsection III-C3), uses another *operation*, or contains multiple *operation steps*. It also contains a set of input and output mappings that use *attribute mappings* (Subsection III-C4) to connect, transform, and specify values between *attributes*, depending on the used *technology interface* or *operation*. Due to this strong coupling to the scope of the *operations*, *operation steps* and *attribute mappings* are not reusable, not assigned a PID, and managed as composites inside an *operation*. *Attributes*, *technology interfaces*, and *operations* on the other hand rely heavily on their reusability and are therefore assigned a PID. *Operation steps* are comparable to function calls on a *technology interface* or another *operation*. They can, however, also be seen as subroutines.

3) Technology Interface: Due to the high effort invested in specifying, testing, and executing a technology, it is desirable to make this work highly reusable. We therefore decided to separate the technologies from the problem-specific operations. Instances of the *Technology Interface* class realize the reusability layer by providing an environment-independent interface to the execution. *Technology interfaces* specify a set of input *attributes*, a set of output *attributes*, and reference a set of Adapter FAIR-DOs via their PIDs.

These Adapter FAIR-DOs are specific to the executing environment and actually implement how the *technology interface* is executed on any given system. In our envisioned approach, the executing systems specify a *type profile* for their *adapters*, which in turn specify machine-interpretable information. This enables the adapter to be downloaded, verified in its integrity, and executed, being subject to the security policies of the

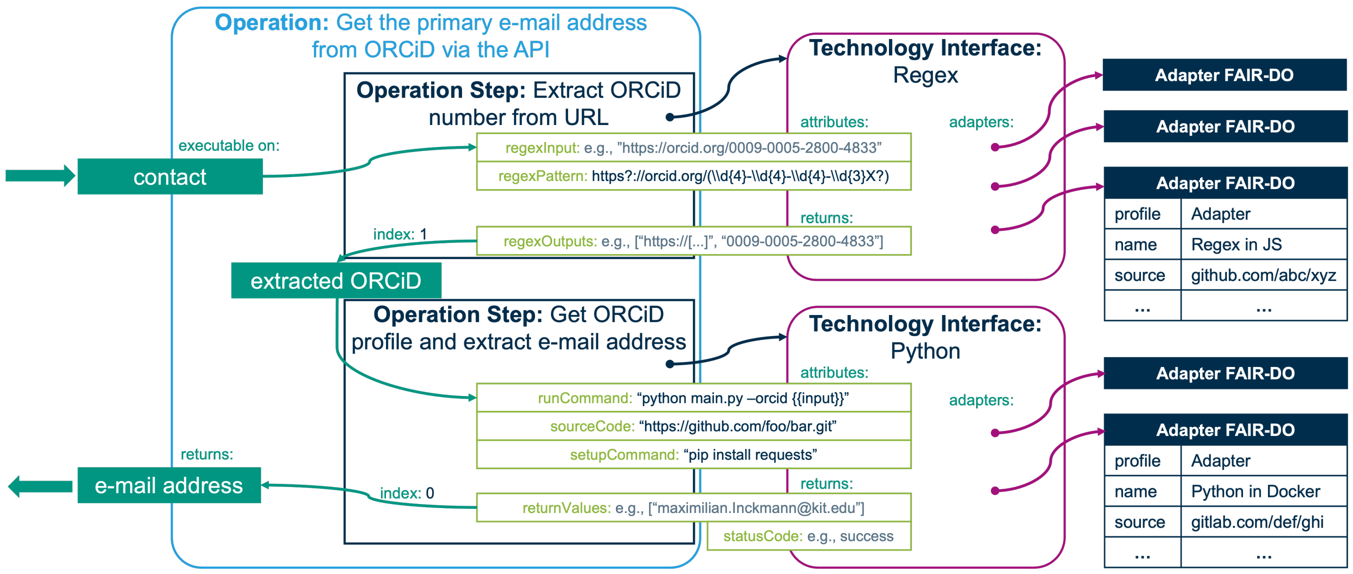
Operation: Get the primary e-mail address from ORCiD via the API
Operation Step: Extract ORCiD number from URL
executable on:
contact
regexInput: e.g., "https://orcid.org/0009-0005-2800-4833"
regexPattern: https?://orcid.org/(\\d{4}-\\d{4}-\\d{4}-\\d{3}X?)
index: 1
regexOutputs: e.g., ["https://[...]", "0009-0005-2800-4833"]
extracted ORCiD
Operation Step: Get ORCiD profile and extract e-mail address
runCommand: "python main.py –orcid {{input}}"
sourceCode: "https://github.com/foo/bar.git"
setupCommand: "pip install requests"
returns:
e-mail address
index: 0
returnValues: e.g., ["maximilian.Inckmann@kit.edu"]
statusCode: e.g., success
Technology Interface: Regex
attributes:
adapters:
returns:
Technology Interface: Python
attributes:
adapters:
returns:
Adapter FAIR-DO
Adapter FAIR-DO
Adapter FAIR-DO
profile | Adapter
name | Regex in JS
source | github.com/abc/xyz
... | ...
Adapter FAIR-DO
Adapter FAIR-DO
profile | Adapter
name | Python in Docker
source | gitlab.com/def/ghi
... | ...

Fig. 2. Example of a complete FAIR-DO Operation

executing system. We did not model Adapter FAIR-DOs as classes in our model (Figure 1) since they just need to be uni-directionally referenced from the *technology interfaces* and their content might vary between the executing systems which is perfectly solved by *type profiles* in FAIR-DO records that also have a PID.

Relating this to the regular expressions used in our example (Figure 2), we see that there are different libraries and APIs for different environments. In this case, we propose to create a *technology interface* for the "Regex" technology that accepts an input string and a pattern while returning an array of strings for the regex groups (the first element always represents the fully validated string). For this "Regex" technology, we can then implement adapters for multiple environment (e.g., a JavaScript web-browser environment, a Python environment, a Java environment). The executing environment 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 these *attributes*. 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 the *operation* (light blue) demands a "contact" *attribute*, which is then transferred into the "regexInput" *attribute* of the "regex" *technology interface*. Similarly, an element from the output of the "regex" *technology interface* is extracted, transformed to the definition of an *attribute* and used in another *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. Information such as the regex pattern extracting parts of an ORCiD-URL, the source code location of a Python script, or the setup necessary to run the script pertains to the *operation* itself rather than to 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 down-cast 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 down- and up-casting, the executing system needs to validate the input values for the *attribute mappings* against the *data type* the output *attribute* complies to.
- **Addressing items in an array:** In Figure 2, there are two examples for the use of up-casting mechanisms, although they do not perform a 1-to-1 mapping, but rather an n-to-1 mapping. The "regexOutput" and "returnValues" *attributes* have a cardinality greater than one, and can therefore be considered as an array of values. However, the "extracted ORCiD" and "e-mail address" *attributes* have a cardinality of exactly one, which necessitates 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.

- **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 graph database Neo4j[10] is used in combination with Spring Data Neo4j[11] and Spring Data REST[12], making IDORIS capable to provide an automatically generated and fully HATEOAS-enabled RESTful-API for CRUD functionality, solely based on our model. More advanced features that demand additional logic, such as resolving the inheritance hierarchy and retrieving available *operations* for *data types*, are exposed using traditional Spring Web MVC endpoints[13].

A. Graph database

Due to the high inter-connectivity 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 technology stack of IDORIS, which allows us to implement our typing model with only minor technical changes, thus enhancing the expressivity of the graph. IDORIS uses these 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, circle detection, and relationship querying provided by Neo4j directly inside our graph database.

B. Rule-based validation and processing

Since our model depends heavily on the correctness of user-provided information, IDORIS must validate this data both, syntactically and semantically. Especially when realizing the inheritance mechanisms for *atomic data types* and *type profiles*, a validation mechanism that is able to act not only on individual entities but also on their contextual relationships is needed. This feature is clearly beyond the capabilities of a JSON schema.

We realized a modular, rule-based approach for validation to enhance its maintainability. This is accomplished by using the "Visitor" design pattern [23] that separates logic from model classes in a highly modular fashion and is therefore often used, among others, for semantic validation and optimization inside 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 of 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, defined our approach for type-associated FAIR-DO Operations in Subsection III-C, and illustrated a running example in Figure 2. In this subsection, we will reuse this running example as a use-case and provide an excerpt of the actual graph data structure in Figure 3 (as described in Subsection IV-A) with all relevant nodes and relations that describe an *operation*. For better readability, we will only show a description of the contents of each node, omitting its properties. The color-labels of the nodes in our labeled-property graph match those of Figures 1 and 2: the *operation* and its *operation steps* are 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 exploited for more complex queries: with respect to our example, the red relations form a circle in our graph, representing the flow of data within the "Get primary e-mail from ORCiD via API"-*operation*. It is visible how the data flows from the "contact" *attribute* through the *attribute mapping* into the "regexInput" *attribute*, that is processed by the "Regex" *technology interface* whose output is then again transformed and inserted into a command starting a "Python script" that outputs the "e-mail address". This circle is queried using the Cypher query:

[7]https://github.com/maximiliani/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-data/rest/reference
[13]https://docs.spring.io/spring-framework/reference/web/webmvc.html

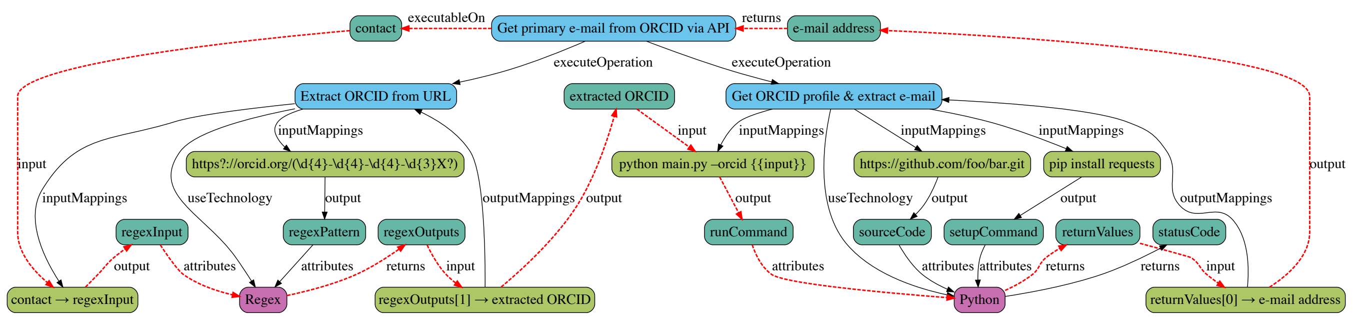

Fig. 3. Excerpt from the Graph representing a FAIR-DO Operation stored in the database

`MATCH (m1) WITH collect(m1) as nodes CALL apoc.nodes.cycles(nodes) YIELD path RETURN path)`. The same query returns an additional circle for each *operation step*, representing levels of abstraction inside an *operation*. This circle detection is useful for a future executing system to automatically parallelize processing and ensuring data is available when it is needed. The concrete use case of this example can be simplified by using a platform-independent execution mechanism, such as Web Assembly (WASM) [24], 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. We do not intend to develop a “universal programming language”, but instead to facilitate the variety of languages, frameworks, and tools that already exist with our technology-agnostic model for FAIR-DO Operations.

IDORIS must also ensure that no invalid cycles are introduced. Examples for such unwanted circular dependencies include, but are not limited to: circles in the inheritance hierarchy, *type profiles* using themselves as *attributes*, and an *operation* calling itself within an *operation step*. To avoid this, we used the path finding algorithms of Neo4j and created a rule for IDORIS’ rule-based validator system (Subsection IV-B). In Listing 1, we show an excerpt of such a validator in Java-like pseudocode that also creates error messages via the API including severity level, message, and the entity of interest. This feature of IDORIS assists users with detailed error messages when creating new elements and ensures data integrity. By implementing separate validator classes for each rule, we enhanced the maintainability of our codebase through separation of concerns. These validators can also be used to ensure semantic correctness, e.g., by ensuring no conflicts exist in the inheritance hierarchies of *atomic data types* and *type profiles*.

```
public class AcyclicityValidator extends Visitor<
    ValidationResult> {
    private final Neo4jClient neo4jClient;

    // Visitor method to ensure an operation does
    not call itself
    public ValidationResult visitOperation (
    Operation operation) {
        return doesNotExecuteItself(operation);
    }

    // Visitor method to ensure it has no recursive
    dependency on itself
    public ValidationResult visitTypeProfile (
    TypeProfile typeProfile) {
        return ValidationResult.combine(
            doesNotInheritFromItself(typeProfile),
            doesNotUseItselfAsAttribute(typeProfile)
        );
    }
    [...]
    private ValidationResult doesNotInheritItself (
    DataType dataType){
        String query = "MATCH path = (n:DataType {
    pid: $nodePID})-[:inheritsFrom*1..]->(n) RETURN
    path LIMIT 1";

        // Query the path from the Neo4j database
        boolean hasCycle = neo4jClient.query(query)
            .bind(dataType.getPID()).to("nodePID")
            .fetch()
            .first()
            .isPresent();


        return hasCycle
            ? new Error("Circular inheritance
    detected", dataType)
            : new OK();
    }
}
```

Listing 1. Excerpt of the Acyclicity Validator role in pseudocode

Figure 4 shows detailed examples of two application cases of *Type Profiles* (described in Subsection 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 FAIR-DOs that adhere to it. One of these *attributes* (Subsection 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* (Subsection 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

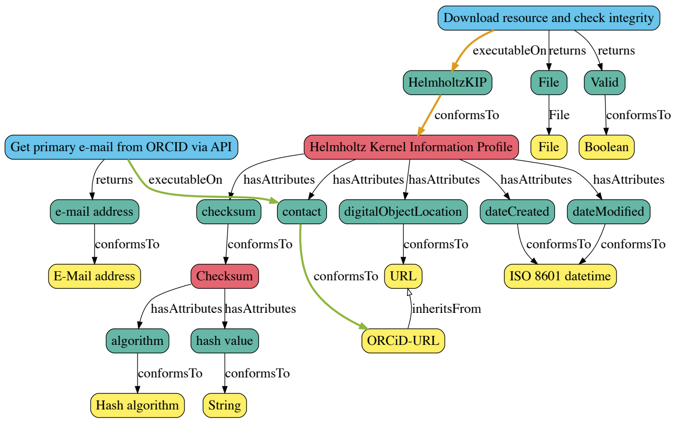

Fig. 4. Excerpt from the Graph representing Data Types in the graph database

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 Section II). 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 decided to not adopt the duck typing variant of "Attribute Typing" by allowing only exactly one *attribute* an *operation* is executable on. 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

We designed our integrated typing model based on the concepts and development of the ePIC and EOSC DTRs [9]. The core concepts — defining simple value syntax 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 (Subsection III-A1). To define complex structures, our model combines the *PID-InfoTypes* (for complex JSON values inside a FAIR-DO) and the *KernelInformationProfiles* (for the structure of FAIR-DOs) with the *Type Profile* class (Subsection III-A2). As a new approach, *Atomic Data Types* and *Type Profiles* themselves are abstracted by the *Data Type* class, which reduces redundancies and provides a strong definition of the term "data type" within our model, enhancing its semantic clarity. The newly introduced inheritance mechanisms for both, *Atomic Data Types* and *Type Profiles* promote reusability of their instances and already allows for basic polymorphic behavior through subtyping, facilitating reuse of the association between *data types* and *operations*. This 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. Unlike *PID-BasicInfoTypes*, our *Atomic Data Types* do not contain fields for specifying measurement units or categories, and exclude other rarely used or undocumented fields, to enhance the semantic clarity of the typing model. For the same reason, the former *SubSchemaRelation* for PITs and KIPs [9] was split into a validation policy and a flag indicating whether additional attributes are allowed.

Unlike the ePIC and EOSC DTRs, IDORIS is not based on JSON schema. 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 way, we can describe type-associated operations and ensure the quality of information stored in IDORIS. We decided against an RDF-based system due to its limited integration into the Spring framework, higher modeling complexity with triples, and steeper learning curve. Although RDF-based graph databases, such as Apache Jena[14], offer potential advantages, including an 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 integrated model significantly 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 upon the models used in existing DTRs by integrating inheritance mechanisms, streamlining the components of the type system, and providing mechanisms that associate data types to 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.

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

References

[1] Directorate-General for Research and Innovation (European Commission), 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," 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 (BigData)*. 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] 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/

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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

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

[16] 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

[17] 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

[18] 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

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

[20] 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

[21] 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

[22] 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

[23] 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.

[24] 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/