International FAIR Digital Objects Implementation Summit 2024
Full Papers
https://doi.org/10.52825/ocp.v5i.1392

Published: 18 Mar. 2025

How to Design FDO Profiles and Kernel Attributes?
An Investigation Along Processing, Grammars and Automata

Ulrich Schwardmann[1,*]

[1]Gesellschaft für wissenschaftliche Datenverarbeitung Göttingen, Germany
*Correspondence: Ulrich Schwardmann, uschwar1@gwdg.de

Abstract. A major goal for FAIR Digital Objects (FDOs) is to enable machine readability, interpretability and actionability for data and metadata of digital objects. This paper examines, how this can be achieved at the level of processing, grammars, derived languages and push-down automata, and describes formal requirements for the definition of FDO records.

After a description of FDOs and what requirements are necessary that they can be processed by machines the importance of types for the processing of bitstreams as well as key-value-pairs in attributes is highlighted and the different options to represent values in attributes of FDOs are investigated.

Machine actionability describes the knowledge of machines about how to process the object. This requires to process not just the languages that can be used for attribute definitions to determine the type of the value, but also the grammars that generate these languages. In this paper a generic way is investigated, how machines can read languages for grammar rules and apply them. Also a way out of a possibly unlimited need for machines able to process different such description languages is presented.

Even if these findings are on an abstract level, the outcome has direct consequences for the way, how FDO records, data types or profiles have to be defined, which standardization agreements are needed and thus how attributes, types or profiles have to be implemented. It can be seen as a theoretical guideline for standarization and for implementation.

Keywords: FAIR Digital Objects, Types, JSON, Context Free Grammars, Automata.

1. FDOs for Machines

The FDO-Forum describes FAIR Digital Objects (FDOs) as machine actionable units of information bundling all information that is needed to enable FAIR processing of any included bit-sequence. Based on the FDO Forum FDO Requirement Specifications [1] an FDO needs to fulfill the following simplified requirements:

- A PID, standing for a globally unique, persistent and resolvable identifier, is assumed to be at the basis for FDOs.

- This PID resolves to a structured FDO-Record compliant with a specified FDO-Profile which leads to predictive resolution results.
- The FDO-Record needs to contain mandatory FDO (kernel) attributes, may contain optional FDO Attributes and attributes agreed upon and defined by recognized communities.
- For generic FDOs the reference to an FDO-Profile is mandatory.
- For FDOs refering to data additional attributes are mandatory:
  - reference to the bit-sequence(s) encoding data,
  - references to the different metadata resources and
  - an FDO-Content-Type.

A type of an object is understood here as a description on how to process the object. Along the characterization of Parnas, Shore and Weiss [2] a type can be provided by a representation of the value space, exactly described by successively more primitive types, possibly augmented with operations on elements of that value space or less exactly described, partly encapsulated value spaces. The FDO-Content-Type is therefore the description how to process the content: the bit sequence referenced in the FDO record. An FDO-Type is assumed to be machine readable.

The FDO-Profile is the type of the FDO-Record itself. The type of the attributes or the profile is given by attribute or profile definitions. They constrain the value space of them, but also for instance the description language (JSON/XML etc.) used or possible applicable operations. This way it decribes, how to process them. These are special aspects of the following dependencies around FDOs, as described in a paper of the TSIG WG of the FDO Forum [3].

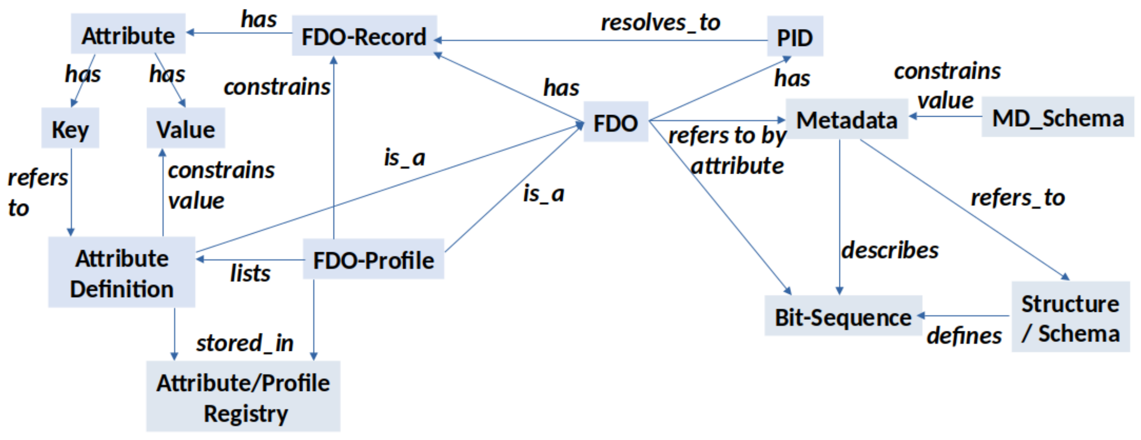

Figure 1. *An FDO has a record of key-value-pairs determined by a profile that is an FDO of its own.*

As a result the definition of FDOs is recursive, because the profile, types and the references are given as attributes in the FDO record and the profile as well as attribute and type definitions are integral parts of the definition of FDOs and are again FDOs. The processing therefore will also be recursive.

1.1 Processing an FDO Record

From the user's perspective the processing of an FDO record starts with the resolution of a PID/URI into a PID/FDO record and, according to a given profile, this record is parsed into a set of attributes consisting of key-value-pairs.

1.1.1 The Processing Loop

After an identification of the attributes of relevance in the FDO the key reference and the definition of them are resolved and parsed and all values of interest are processed. In the case all values are in the FDO Record, the loop terminates. Otherwise if they are referenced by an ID/URI, it is necessary to recurse in the loop.

This processing loop, as described by Christophe Blanchi in an unpublished draft of the TSIG working group of the FDO Forum [4], therefore is an iteration of answers to the questions "where to find and how to process an object?", in other words for location and type. This obviously only works, if a resolver from identifiers to FDO records or terminal bit sequences exists.

The typical approach for processing a single type of objects is to build a machine suitable to process objects with that given type (i.e. a schema or MIME type). This approach assumes that both, location and type of the object, are a priori given. Usually this is realized by a reference to the object and the creation of a specific machine for the object type at hand.

Location and type of FDOs are required here at each level of the FDO processing loop above in a similar way. It starts with the PID pointing to the FDO record with its profile and continues with the attributes and their definitions. The object location is given by reference. And since the objects often are not just of primitive type, their description is then also given by reference with possibly different description languages.

An approach with specificly created machines at each state of the loop therefore will not work here. In the processing loop an automation to gather these requirements, location and type, is needed and this requires a generic approach and some standards, especially for the type descriptions, that can be handled by machines.

1.1.2 Types and Type Definitions for FDOs

For attributes as key-value-pairs there is a helpful and simple predefined structure given by the agreement that the key describes the type of the object given by the value. Such a dependency structure is often used in the context of metadata or for key-value structures as such.

This agreement can also be made for FDOs. Here the key in attributes is the type of the value and this type is always a reference to another FDO that describes how to process the value as its object. This referenced FDO is called type definition.

This is the case for all attributes given as key-value-pairs in FDOs. In other words attribute definitions are FDOs that provide the type of possible attribute values and their PIDs are used as keys in attributes of FDO records.

But the allowed value space defined by the type in the key can be complex, such that a direct storage of the value in the FDO record is not always possible. Moreover the exact value space might not be finally known at the time a profile is defined.

For instance in a first step of automation the type of a bitstream could be sufficiently described by a MIME-type which can be given by just a string from an enumeration list. But also it should be possible that a type definition can be a complex structure able to make descriptions in different description languages like Parquet [5] Avro [6] or even currently still unknown concepts.

Therefore the introduction of just one special key that provides the type of the bitstream of an FDO will not be sufficient in general and another level of redirection might be necessary to cover different description concepts for types.

This raises the question how to distinguish in an attribute, whether the value is either the object itself, for instance just a number, a string or a simple JSON object, or the reference to the object or even a reference to another attribute in the FDO record that finally defines the type and what the criteria for an automated decision on that might be.

1.1.3 Distinction between Inline or Internally and Externally Referenced Objects

A possible criterion for a discrimination between value and reference could be that, if the value of an attribute matches the type, it can be assumed that this matching is intentional. In this case the expected value could be assumed to be only for the further processing of external clients. However this criterion is only heuristic and has limitations by possible unsufficiently sharp type definitions that do not discriminate unwanted references. Furthermore it is expensive, because it requires to determine the value space and to validate the value.

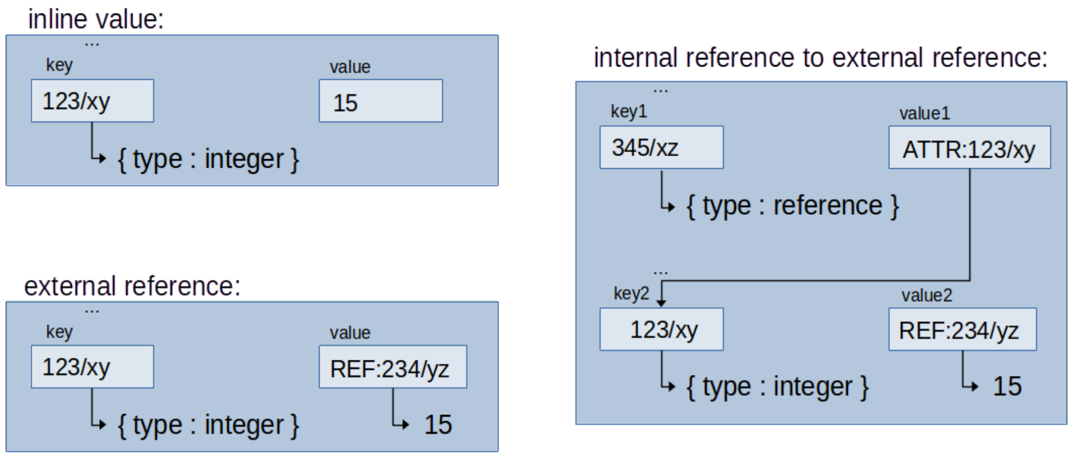

Figure 2. *References given by values need additional flags to distinguish internal and external targets.*

The discrimination between external value reference and internal type reference could be given by the existence of the reference as additional key in the FDO record. This criterion is not expensive, because it can be applied directly on the FDO record, but it is also only heuristic. Cases could exist, where a reference is meant as an external value, for instance an attribute type, and at the same time this reference is also given as key of an additional attribute with according value.

A much better way to distinguish such cases is to flag the value of the attribute somehow, for instance by prepending the reference strings with a characterising prefix, like "FDO-REF:" or "FDO-ATTR:" and at the same time exclude such a prefix from all values that are not references. But this needs a standardization decision by the FDO-Forum.

1.2 Normalisation of the FDO record

An FDO is given only by a reference, the PID, that resolves to an FDO record. Usually there is no key-value-pair available with that PID as a value, where the key can lead to its type.

1.2.1 A Normalisation Operation and 0.Profile

Therefore this requires first of all an agreement on some basic structure for the type of a generic FDO record and then a normalisation operation from the actual, implementation dependend FDO record to that generic FDO record. Its type could be described as an object of key-value-pairs with some minimal attributes as needed.

In accordance to the FDO requirements for this type of a normalized FDO record a two step approach is proposed. In a first step a specific key for the profile is assumed to be in the FDO record, i.e. the **0.Profile** key, that has as value the profile, given directly as object or as a reference. The 0.Profile as key refers to a type for possible profiles, such that the profile as value of the attribute can be read by machines.

Variations between profile structures, dependent on the implementation, are possible. These variations could be given as allowed alternatives in the type definition of 0.Profile. Mappings to a single generic 0.Profile structure are an alternative, which might be advantageous, because this can also be implemented as part of the normalisation operation for the FDO record.

As a result the profile is an object of key-value-pairs that describes the structure of the normalized FDO record, i.e. it determines, which attributes can be expected in the FDO record, which are mandatory or optional, which names can be used as an alias for the attribute definition behind the attribute that provides the type of the attribute. At its minimum it requires only the 0.Profile key for the FDO record, but in general it requires different more complex structures for different classes, sometimes called genres, of FDOs. In other words the profile is the type of the FDO record as an extension beyond the always provided profile attribute.

1.2.2 The Profile for Data FDOs

As described above there is an agreement in the FDO requirement specifications that the FDO record of data objects in particular requires a reference to the bit-sequence(s) that encode the data, the FDO-Content-Type of the bit-sequence(s), references to the different metadata resources and as always the reference to the FDO-Profile.

For the FDO record there are two possible options how to organize the reference to the bit-sequence and the FDO-Content-Type according to the general approach that the key refers to the type of the object.

If a special attribute key, i.e. **URL** or some other reference to the payload is used as key for references to digital objects, its type definition allows all possible, also repository internal references to objects, that in general cannot be recognized as references by their form. Therefore this key has to have a value as reference by definition and the key cannot wear the type definition of the digital object behind the reference as value.

Therefore it is suggested to provide an additional attribute for the object type with an additional special key, i.e. **0.ContentTypeRef**. The attribute with this key can be one of the cases above with value or internal or external reference. This allows sufficient flexibility to cover the simple MIME-type case, where the value is a string out of a predefined enumeration list of possible known types. But it also could be a PID reference to a more complex content type structure. And it could be a reference to another attribute in the FDO record that finally provides the type definition. This way it would be possible to start the implementation with MIME types and later it is possible to extend the concept to more complex content type descriptions.

2. Grammars for FDO Records

We have described above that the keys in FDO Records are PIDs referencing types by other FDOs. These types as referenced FDOs are called attribute definitions that describe what can be expected in an attribute and determine the type and value space of the value in key-value-pairs.

But a machine still needs to find out, how to process the type or attribute definition behind the key. This is a matter of the language a machine has to process and therefore the context free grammar (CFG) behind that language. How these grammars should be chosen, will be discussed in this chapter.

2.1 How to Process Types

Types have to be expressed in a language that a machine can process, which implies that there is a CFG behind this language used in the type definition.

Because types or attribute definitions are themselves FDOs of data objects, they have an FDO record that contains beside the reference to the type definition itself an attribute with the FDO-Content-Type: the type of the type. All necessary information seems to be available for machines this way. However there are three problems with this approach.

Explosion of Types Problem: First of all this approach has to maintain a lot of references and therefore redirections between the registries, where all these definitions are stored. This in practice can be mitigated by caching these definitions, because this leads to simple pointers inside the server code. A necessary condition for this approach is however that the number of types is limited. But an explosion of types can easily happen, if there are no rules and processes that strongly encourage the reuse of existing types.

Infinite Type Definition Problem: The other, more fundamental problem is that each type itself needs a type, a description, how to process it, that again needs a type, which leads to an infinite definition of types. This infinite definition problem can be solved only, if the way how a definition is described at a certain step in the chain equals that of the former step. Occasionally this is possible such that a solution can be found. For instance context free grammars (CFGs) can be given in Backus-Naur form and the Backus-Naur form can be described as CFG. So the grammar and the grammar definition can be given by the same grammar. This example of the Backus-Naur form actually gives the hint, how the solution of this infinite definition problem can also be given on the level of descriptions for types or attribute defintions.

Focussed Specification Problem: However a third problem pops up with this possible solution for the second problem. Automated processing becomes the less precise and the less focussed the more general the grammar rules are. And Backus-Naur forms are by definition as general as possible for automata. Type definitions on the other hand need to provide very specific descriptions of the type at hand. Therefore the generic result of Backus-Naur forms on alphabets and grammars is not sufficient for machines to process objects along specific information given by types. It is therefore necessary to find more focused subsets of grammars that solve the infinite definition problem, but are precise enough to allow machines to make decisions on how to process objects.

Which kind of grammar is best suited for specific machine decisions and at the same time solving the infinite definition problem will be investigated in the following sections. The possible explosion of types is then covered in the last section of this chapter.

2.2 Machine Readability, Interpretability and Actionability

In the specification of the FDO-Forum on FDO Machine Actionability [7] three levels have been stated from machine readability over interpretability to actionability.

Machine readability is obviously an activity of automata and is understood here as the acceptance of the language of the FDO presentation by some automaton.

For the interpretability assumption this is also true but not directly obvious. Elements are called interpretable in that paper, if their meaning has been specified in some semantic artifact as augmented assertions that build lists of entity relations. These relations are described in a structured way, given for instance in RDF or JSON-LD. Thus also for machine interpretability in the sense above the relations are structured into languages that are accepted by automata and based on CFGs.

Machine actionability here is the knowledge, how an automaton reads and processes a digital object. This knowledge is represented as the type of the object. For machine actionability this represention needs to be structured into languages that become accepted by automata.

A fundamental theorem of automata theory states that if a language is accepted by a push down automaton, the language is context free and conversely that for a context free language there exists a push down automaton that accepts this language [8].

For machine actionability this means that it is not sufficient to know, how the language is described that is used for the construction of such types. The machine needs rather to be able to process the grammar itsself, i.e. its production rules. There must be a meta grammar, such that the machine can read the production rules of the grammar. Therefore machine actionability is a kind of a second order machine readability. And this is a recursive demand, because as shown before there might be an infite definition problem for types.

But alltogether in this model of readability over interpretability to actionability everything is described as syntactical processing and can be described by grammars and automata. Therefore the model is essentially inline with the observation of L. Floridi [9].

2.3 A Grammar Based on Operator Production Rules

The goal here is to understand, how a grammar can be refined in a way that it enables machines to make focussed processing, but at the same time without loosing its capacity to describe its own grammar as it is possible for instance with Backus-Naur forms. In a first step it is therefore helpful to have a look at a particularly simple form of grammar descriptions, the Greibach Normal Form (GNF)[10] with the resulting expressions of production rules of the form: $\mathcal{V}_i \rightarrow \odot_i \mathcal{V}_{i,1} \ldots \mathcal{V}_{i,k_i}$. It is possible to transform an arbitrary CFG into this Greibach Normal Form without changing the language described by the grammar with production rules.

Formally the outcome of GNF production rules is a single terminal followed by a sequence of non-terminals. The leading terminals of the production outcomes could be interpreted as operators on the sequence of non-terminals as the operands. Because this interpretation as grammar with operator production rules is used later, it is put in the foreground here. Since the Greibach normal form is particularly simple, this leads to the following results on grammars that produce grammars:

Proposition 1: Operator production rules can be generated by a grammar based on non-terminals $\{\mathcal{D}, \mathcal{L}, \mathcal{O}, \mathcal{S}\}$, with startsymbol $\mathcal{S}$, a possibly infinite set of terminals $\{\Rightarrow, \odot_i, \mathcal{V}_i\}$ with a range of indexed symbols for operators, non-terminals and terminals in a described grammar and production rules:

$$\begin{aligned} \mathcal{S} \rightarrow &\quad \mathcal{V}_i \mathcal{D} \\ \mathcal{D} \rightarrow &\quad \Rightarrow \mathcal{O} \\ \mathcal{O} \rightarrow &\quad \odot_i \mathcal{L} \\ \mathcal{L} \rightarrow &\quad \mathcal{V}_i \mid \mathcal{V}_i \mathcal{L} \end{aligned}$$

In the second production rule are two arrows involved. The first arrow is as usual the grammar rule relation. The second double arrow is in this grammar just a character that furthermore is a placeholder for the intended character $\rightarrow$, only introduced here to distinguish between the grammar rule arrow and the character for the grammar rule arrow.

The first three rules, introduced here to fulfill together the Greibach normal form, can be summarized in a non-Greibach form to $\mathcal{S} \rightarrow \mathcal{V}_i \Rightarrow \odot_i \mathcal{L}$. The last rule generates a chain of symbols $\mathcal{V}_{i,1} \ldots \mathcal{V}_{i,k_i}$ used as variables in the generated grammar, such that the grammar produces $\mathcal{S} \rightarrow \mathcal{V}_i \Rightarrow \odot_i \mathcal{V}_{i,1} \ldots \mathcal{V}_{i,k_i}$ The variable and the operator symbols here can vary across all elements $\odot_i$ and $\mathcal{V}_i$ in its terminal sets.

The equivalent grammar given by replacing the double arrow character with the single arrow character is then able to generate production rules of the form $\mathcal{V}_i \rightarrow \odot_i \mathcal{V}_{i,1} \ldots \mathcal{V}_{i,k_i}$ that are part of the original grammar in GNF above, such that the start symbol produces a string that is interpreted as rule in the language of this generated grammar.

Since all production rules in the described grammar are in Greibach normal form, the following holds:

Proposition 2: A grammar that generates operator production rules can again be based on operator production rules.

A machine based on this grammar can read the production rules above and those of its grammar itself. This is a similarly generic result as the fact that machines based on Backus-Naur forms can read Backus-Naur forms. Such a grammar has obviously production rules of a much simpler structure then Backus-Naur, but still can generate grammars for all possible context free languages.

Its language is based on the terminals in the generating grammar as indexed symbols for operators, non-terminals and terminals in the described grammar, which leads to:

Table 1. *An operator grammar for JSON*

Type Definitions :	$\mathcal{T} \rightarrow$	$\mathcal{O} \mid \mathcal{A} \mid \mathcal{B} \mid \mathcal{C}$
Combined Definitions :	$\mathcal{C} \rightarrow$	$\mathcal{T}_1 + \ldots + \mathcal{T}_k, k \geq 1$
Object Definitions :	$\mathcal{O} \rightarrow$	$\{\mathcal{P}_1, \ldots, \mathcal{P}_k\}, k \geq 1$
Key − Value − Pair Definitions :	$\mathcal{P} \rightarrow$	$[l, \mathcal{T}, \mathcal{Q}], l \in \text{String}$
Array Definitions :	$\mathcal{A} \rightarrow$	$[\mathcal{T}_1, \ldots, \mathcal{T}_k], k \geq 1$
Basic Definitions :	$\mathcal{B} \rightarrow$	Null | Boolean | String | Number
Quantifiers :	$\mathcal{Q} \rightarrow$	? |!

Proposition 3: The complexity of a grammar given by operator production rules is only determined by the terminals of its generating grammar.

2.4 A Grammar for JSON Terms

Types and attributes definitions describe, how to process the object in question, for instance for bit sequences. For processing the description of bit sequences can be the schema of the data stream given as digital objects in some abstract language like Java for Parquet [5] or JSON for Avro [6].

Also the description for processing FDO records, attribute definitions and profiles can be provided as elements in JSON, as it is done for instance in the implementation of type registries in CORDRA [11] that are based on the data model described as an outcome of the RDA WG on Data Type Registries [12].

These type and attribute definitions are provided in a language of a CFG that describes certain JSON records based on subsets of primitive types as terminals and variables produced from arrays or objects. Other description languages (like XML, ...) could be used here as well, but for simplicity and comparability reasons a restriction on JSON is assumed in the following. Therefore a good starting point to find a more focussed, but still self describing grammar for types and attribute definitions is to examine a description of JSON by a grammar.

The description of JSON as a CFG in Table 1 is a modification of the grammar for JSON types given by Baazizi e.a. in [13].

The variable for key-value-pair definitions consisting of triples describes objects of keys, values and an obligation as third element in the triple. These triples are collected to objects, which are sets with the additional property that only one element per key is allowed.

One important observation here is that the language of this grammar describes not just words in terminals but sets, because the final terminals of the grammar, like strings or numbers in the Basic Definitions are themselves sets. These sets can be refined to subsets. For numbers for instance to intervals or multiplicities for strings by regular expression. These language refinements can be given by additional rules in the grammar.

Another observation is that in contrast to the description of Baazisi e.a. each production rule here fulfills the requirements of an operator grammar, because all expressions are given by operators applied to a list of variables, where the operators are alternatives (|), unions ($+$), tuples ($\{\ldots\}$) or lists ($[\ldots]$).

Table 2. *A self describing operator grammar for JSON*

Start Definition :	$\mathcal{S} \rightarrow$	$\mathcal{T} \mid \mathcal{R}$
Grammar Rule Definitions :	$\mathcal{R} \rightarrow$	$\mathcal{V}_i \Rightarrow \odot_i \mathcal{L}$
Type Definitions :	$\mathcal{T} \rightarrow$	$\mathcal{O} \mid \mathcal{A} \mid \mathcal{B} \mid \mathcal{C}$
Combined Definitions :	$\mathcal{C} \rightarrow$	$\mathcal{T}_1 + \ldots + \mathcal{T}_k, k \geq 1$
Object Definitions :	$\mathcal{O} \rightarrow$	$\{\mathcal{P}_1, \ldots, \mathcal{P}_k\}, k \geq 1$
Key − Value − Pair Definitions :	$\mathcal{P} \rightarrow$	$[l, \mathcal{T}, \mathcal{Q}], l \in \text{String}$
Array Definitions :	$\mathcal{A} \rightarrow$	$[\mathcal{T}_1, \ldots, \mathcal{T}_k], k \geq 1$
Basic Definitions :	$\mathcal{B} \rightarrow$	Null | Boolean | String | Number
Quantifiers :	$\mathcal{Q} \rightarrow$	? | !
Variable String :	$\mathcal{L} \rightarrow$	$\mathcal{V}_i \mid \mathcal{V}_i \mathcal{L}$
Grammar Variable Definitions :	$\mathcal{V}_i \rightarrow$	$\{\mathcal{T}, \mathcal{C}, \mathcal{R}, \mathcal{O}, \mathcal{P}, \mathcal{A}, \mathcal{B}, \mathcal{Q}, \mathcal{V}_i, \odot_i\} \subset \text{String}$
Grammar Operator Definitions :	$\odot_i \rightarrow$	$\{\mid, [\ldots], \{\ldots\}, +\}$

This grammar for JSON is still not quite sufficient to allow the description of the required kind of production rules that then enables the grammar to also describe its own grammar production rules.

2.4.1 Bootstrapping with a Binding Operator on an Array

However it is possible now to combine the self describing operator based GNF grammars from Proposition 1 with the JSON grammar in GNF above by regarding the production rules of this grammar as operator production rules of the form $\mathcal{V}_i \Rightarrow \odot_i \mathcal{V}_{i,1} \ldots \mathcal{V}_{i,k_i}$. For this the arrow character needs to be exchanged again and an additional production has to be introduced that selects the necessary characters from the set of characters used for the variables of the JSON grammar. The same needs to be done for the operators used there to fulfil the requirements for Proposition 1.

Such a grammar can then be enhanced with the additional rules in GNF that describe the operator production, summarized as $\mathcal{S} \rightarrow \mathcal{V}_i \Rightarrow \odot_i \mathcal{V}_{i,1} \ldots \mathcal{V}_{i,k_i}$, alltogether leading to the production rules given in Table 2.

This new grammar still generates JSON, but after an exchange of the arrow character it also includes expressions of the form $\mathcal{V}_i \rightarrow \odot_i \mathcal{V}_{i,1} \ldots \mathcal{V}_{i,k_i}$ that are able to describe the grammar production rules, but do not belong to the JSON language itself.

A self describing grammar that stays completely inside JSON can be obtained however with a small loss of its expressiveness for the JSON language. The grammar rule production $\mathcal{R} \rightarrow \mathcal{V}_i \Rightarrow \odot_i \mathcal{L}$ above describes the replacement of a variable in the JSON grammar by an operator in the JSON grammar followed by a list of JSON objects.

Such a construction can also be easily mapped to JSON itself using an object like $\{[\text{var}, \mathcal{V}_i, !], [\text{opr}, \odot, !], [\text{ops}, \mathcal{L}, !]\}$ that consists of three mandatory key value pairs, one for the variable to replace, one for the chosen operation and the last for the list of JSON objects that also can be provided by a JSON array. Similar to the decomposition for JSON objects this expression can be decomposed into operator based rules.

Such a JSON compatible grammar description replaces the grammar rule definition above with the production rule $\mathcal{R} \rightarrow \{[\text{var}, \mathcal{V}_i, !], [\text{opr}, \odot, !], [\text{ops}, \mathcal{L}, !]\}$ and the variable string with an array production $\mathcal{L} \rightarrow [\mathcal{V}_1, \ldots, \mathcal{V}_k], k \geq 1$. Additionally var, opr and ops then need to be removed from the allowed key strings for key-value-pair definitions.

This grammar has a slightly smaller value space for allowed JSON expressions. On the other hand the whole grammar is self describing within the JSON language itself.

For practical purposes the advantage to stay completely inside the JSON language and also to be able to describe rules that generate all used expressions inside this framework will often be crucial and this small lack of expressiveness within the JSON language is often neglectable, because only a part of the possible keys in objects is used anyway.

A JSON grammar with such a production rule definition provides a universal way to make JSON grammar descriptions machine readable (again with this minor lack of expressiveness). An immediate consequence is that also the description for JSON grammar descriptions is readable by machines this way, because the grammar above is itself an operator grammar, which solves the infinite definition problem and allows the bootstrapping.

2.5 Simplified Production Rules for FDOs and Types

For the description of FDO records, attribute definitions and profiles, but also for a bitstream structure or a schema of a structured record usually only specific JSON expressions are used. These expressions are given by a restriction of the allowed key strings in JSON objects as well as by subsets of primitive types as terminals and variables produced from arrays or objects. Because here the value space of possible JSON expressions is much smaller, the minor reduction of the expressiveness of JSON doesn't matter at all and a more focussed processing of these structures is possible for machines,

One of the goals of the data model in [12] introduced by the RDA WG on Data Type Registries was exactly to describe, how a set of attributes could look like that is sufficient for the description of data types.

But binding operators, the essential element to express, how the variables in the list are coupled together, have been missing in this data model. The same is by the way also the case in many metadata models described in tables with dependencies given by hierarchical numbers, like the Dublin Core or DataCite models. All these models are not sufficient for machine actionability in the sense as described above.

This required rule for binding operators for lists of data types have been first introduced and discussed in a previous paper of the author [14], which was the basis of the ePIC DTR [15]. Only with these rules on binding operators for lists of subtypes types, type definitions and types of type definitions become machine readable and therefore the objects machine actionable.

2.6 Identifiers for Production Rules and Data Type Registries

So far the definition of attributes and types is based on a specified grammar based on variables and the usual way is to create automata that are able to read and process the language of this grammar with a local implementation of the variable references.

The DTR WG of RDA had an additional goal in mind with the introduction of data type registries for which its data model (s.a.). The idea was to define and use types and attributes in a global way and to introduce global identifiers that are references to and part of the attribute definitions. For these references an external resolution process was a precondition, provided by a special automaton called PID resolver.

As a consequence attribute and type definitions become globally machine readable. But the global identifiers itself do not prevent multiple definitions of the same

Table 3. *A grammar producing a rule database with identifiers as an extension of an operator grammar*

PID Resolution :	$\mathcal{R} \to$	$p_3(\text{search}(\mathcal{I}))$
Database Update :	$\mathcal{D} \to$	$\mathcal{D} \cup \{\mathcal{E}\}$
PID Expression :	$\mathcal{E} \to$	$[\mathcal{I}, s, \mathcal{T}_j]$
Unique Identifier :	$\mathcal{I} \to$	$\mathcal{I} + 1 \in \text{Number}$
Name as String :	$s \to$	$\text{str} \in \text{String}$
Identifier Init :	$\mathcal{I} \to$	$0 \in \text{Number}$
Database Init :	$\mathcal{D} \to$	$\emptyset$
Operator Productions :	$\mathcal{T}_i \to$	$\odot_i \mathcal{R}_{i,1} \ldots \mathcal{R}_{i,k_i}$

types and in order to avoid an explosion of types by redefining the same type in different environments the reuse of existing types or schemas for metadata is desired.

Production rules in ordinary grammar descriptions are placeholders or references for rules that are unique inside the set of rules. However these placeholders are only used as implicit references inside the grammar description. A reuse of production rules between grammar descriptions needs explicit references that are part of the grammar to ensure self descriptiveness.

With the integration of identifiers and their resolution into the grammar the rules can be expressed as rules on these identifiers, and if the identifiers are globally unique, the rules of grammars can be given in a decentralized way. An automaton that works on this kind of grammar expressions, on the language describing the grammar based on identifiers, as it is the case for types, can work on global and decentralized presented grammar rules.

Since global machine actionability requires a (syntactical) processing of the grammar, the references above have to become part of the grammar. This can be described as an additional addon to generic grammar descriptions, for simplicity reasons in Table 3 shown for Greibach normal forms or operator production grammars.

These identifiers become then ubiquitous in the grammar description. They reference all incomplete parse trees (parse trees with also non-terminals as leaves) including therefore all production rules and each single node inside any parse tree of the grammar in a globally unique way .

The productions of the underlying grammar are described as above as production in Greibach normal form. This grammar stores all expressions available by the operator productions together with a unique identifier and a name for semantical bindings into a simple database, where the triples can be retrieved with a search function on the identifier and the expression by a projection on the third entry in the triple.

In order to implement a global and decentralized automaton that reads this grammar, the numbers needs to be separated into different number spaces for instance by requiring certain prefixes for the numbers used in a local implementation. This requires then also the implementation of a global search engine for identifiers.

An application of production rules on non-terminals in existing identified expressions or incomplete parse trees leads to new, possibly incomplete, parse trees, which together with a new identifer can be stored in the database.

Consequently also a global reuse of existing and identified type and attribute definitions works on a machine level, if the identifier references and their resolution are integrated into the grammar description of attribute definitions.

2.6.1 Grafting in Grammars with Identifiers

Instead of production rules complete existing and identified, possibly incomplete, parse trees can be applied here. This *grafting* of trees is possible if the root variable of the grafted tree, here an identifier, is the same as the identifier representing the variable at the leaves of the incomplete parse trees that becomes extended by grafting.

Grafting is recursive: the building of deeper structures of incomplete parse trees can be repeated until all leaves are (identified) terminals. Parse trees can then be defined from incomplete parse trees, where for each identified non-terminal at the leaves of the tree a new parse tree (or production rule) is chosen starting with that non-terminal.

This process provides an additional unambiguous labelling of parse trees beside the usually used ambiguous labelling by non-terminals. If the grammar itself is unambiguous, the references of parse trees can also be used for each word in the language of its underlying grammar as an additional consequence. For ambiguous grammars equal words may have different identifiers.

From the viewpoint of the context free language these additional non-terminals for the production of identifiers and their resolution are useless symbols that usually would be deleted for simplification of the grammar. Here however they are exactly the additional ingredients that make the parse trees based on identifiers readable by globally acting machines via the resolution.

If terminals, non-terminals and operators are provided with identifiers, the production rules can be represented as collections based on identifiers, as it was suggested in the RDA Recommendation on Research Data Collections [16], and the grammar itself becomes then a collection of collections.

2.7 Self Describing Structures in Existing Attribute Registries

Beside the data model of "PID-InfoType" and "PID-BasicInfoType" a slightly smaller data model for the definition of FDO types was introduced in a new ePIC DTR that uses the schema HDL:21.T11969/87efe6353f9d42f690e3.

That the "PID-InfoType" schema has the necessary complexity to describe this similar complex schema can be shown by the description of the FDO-type-definition in HDL:21.T11148/04bc5ec4f8bb489d7962 in the ePIC DTR for testing [17]. In a similar way also the "PID-InfoType" and "PID-BasicInfoType" could be described by "PID-InfoType".

In this DTR the "subSchemaRelation" key in the substructure "representationsAndSemantics" is a necessary extension of the DTR working group data model of RDA to take the role of the operator description. The operands are given by an array of references provided by an attribute with key "properties". And the identifier represents the variable that can be replaced by this operator binding these operands.

Via a REST service with an according entry point at [18] the definition for FDO-Types can be used to derive a schema at HDL:21.T11148/b72cf35b541e2ef79830. This schema then can validate the correctness of objects in the DTR representing FDO types during creation.

This way a machine is able to read the grammar and to process the language for the definition objects in order to finally validate the objects itself. This machine is also able to process the grammar that describes this grammar, because these two grammars are already the same.

A much smaller self describing data model based on the operator grammar at Table 2 is given in the DTR above by HDL:21.T11969/5c211d1611a829aa06c3 as "OperatorGrammarRule".

The definition is based on a schema HDL:21.T11969/2bf0a35f84cd895c511b. "OperatorGrammarRule" can be derived from its own type definition, where list or tuple validation are used on the variables operators are applied on.

3. Conclusion and Outlook

This theoretical approach leads to a couple of open standardization and implementation possibilities, for which a decision is needed. Topics for needed decisions on the implementation of FDOs are provided here in a brief summary together with a suggestion for such a decision. More details can be found in the text above.

- **Key defines value space:** In attributes the key provides a reference to an attribute definition, which is the type description of the value and an FDO.
- **Profile as list of attributes:** A generic basic structure of FDO profiles is given as a list of mandatory or optional attribute definitions that can be augmented by names as aliases in FDO records.
- **O.Profile:** A generic attribute key is chosen that describes this profile consisting of at least one attribute with key 0.Profile and a profile as value.
- **A normalisation operation:** For each FDO implementation a mapping of the FDO record and the FDO profile in a generic basic structure is provided.
- **0.ContentType:** For the type of the bit sequences in data FDOs the additional generic attribute 0.ContentType is chosen.
- **Discrimination of references:** To distinguish between an inline value or a reference to a value according to the type or a reference to another type attribute in the FDO record the prefixes "FDO-REF:" or "FDO-ATTR:" are used in the keys of attributes.
- **Binding operators:** A description of the kind of coupling of sub types is an integral part of recursive type definitions. The attributes that are used to describe operator and operands need to be marked and a minimal set of operators should be provided by all FDO implementations.

Of particular interest is the question of how to avoid an explosion of types. With the global identification a parallel definition of equal types is not necessary and can be avoided by the search function. But similarity of types requires a knowledge about mappings between type definitions, often called crosswalks. This is usually covered by direct mappings between types, which is cumbersome and in many cases insufficient.

In the theoretical approach above types are defined by grammars. It seems that a mapping between languages based on grammars or even directly between grammars could provide a better understanding of the theoretical background of crosswalks in order to find more generic solutions. A focus on a theoretical understanding of these mappings will be of future interest.

Data availability statement

The data supporting the results of section 2.7 are deposited as data set as "Examples of self defining type definitions" in a FAIR-aligned public repository and can be freely accessed under DOI:10.25625/B5SE35.

Competing interests

The author declares that he has no competing interests.

References

[1] I. Anders et al., “Fdo forum fdo requirement specifications,” version 3.0, *Zenodo*, Mar. 2023. DOI: 10.5281/zenodo.7782262. [Online]. Available: https://doi.org/10.5281/zenodo.7782262.

[2] D. L. Parnas, J. E. Shore, and D. Weiss, “Abstract types defined as classes of variables,” *Proceedings of the 1976 conference on Data : Abstraction, definition and structure*, pp. 149–154, 1976. DOI: 10.1145/800237.807133.

[3] C. Blanchi, M. Hellström, L. Lannom, A. Pfeil, U. Schwardmann, and P. Wittenburg, “Implementation of Attributes, Types, Profiles and Registries,” Apr. 2024. DOI: 10.5281/zenodo.7825572. [Online]. Available: https://doi.org/10.5281/zenodo.7825572.

[4] “Fdo forum tsig working group: Machine rules for accessing fdos,” Accessed: Jun. 19, 2024. [Online]. Available: https://docs.google.com/document/d/1oR1_YZSL6OJbZh4ozoDowD8x3G23MshRX5q8hLnFiQM/edit.

[5] Parquet. “Overview,” Accessed: Jun. 19, 2024. [Online]. Available: https://parquet.apache.org/docs/overview/.

[6] Avro. “Documentation,” Accessed: Jun. 19, 2024. [Online]. Available: https://avro.apache.org/docs/.

[7] C. Weiland, S. Islam, D. Broeder, I. Anders, and P. Wittenburg, “Fdo machine actionability,” version 2.2, *Zenodo*, Apr. 2023. DOI: 10.5281/zenodo.7825650. [Online]. Available: https://doi.org/10.5281/zenodo.7825650.

[8] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory*, 1st. 1979, 3 [HMU06], ISBN: 020102988X.

[9] L. Floridi, *The Fourth Revolution - How the Infosphere is Reshaping Human Reality*. Oxford University Press, 2014, p. 266, ISBN: 9780199606726.

[10] S. Greibach, “A new normal-form theorem for context-free phrase structure grammars,” *Journal of the ACM. 12 (1)*, pp. 42–52, 1965. DOI: 10.1145/321250.321254.

[11] CORDRA. “Homepage,” Accessed: Jun. 19, 2024. [Online]. Available: https://cordra.org.

[12] L. Lannom, D. Broeder, and G. Manepalli, “Rda data type registries working group output,” *Zenodo*, Aug. 2018. DOI: 10.15497/A5BCD108-ECC4-41BE-91A7-20112FF77458. [Online]. Available: https://doi.org/10.15497/A5BCD108-ECC4-41BE-91A7-20112FF77458.

[13] M.-A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, “Parametric schema inference for massive json datasets,” *The VLDB Journal 28*, pp. 497–521, 2019. DOI: 10.1007/s00778-018-0532-7.

[14] U. Schwardmann, “Automated schema extraction for PID information types,” in *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, Dec. 2016. DOI: 10.1109/bigdata.2016.7840957. [Online]. Available: https://hdl.handle.net/21.11101/0000-0002-a987-7.

[15] “Epic data type registry: Web interface,” Accessed: Jun. 19, 2024. [Online]. Available: https://dtr-pit.pidconsortium.net.

[16] T. Weigel et al., “Rda research data collections wg recommendations,” version 1.0, *Zenodo*, Dec. 2018. DOI: 10.15497/RDA00022. [Online]. Available: https://doi.org/10.15497/RDA00022.

[17] “Epic data type registry for testing: Web interface,” Accessed: Jun. 19, 2024. [Online]. Available: https://dtr-test.pidconsortium.net.

[18] “Epic type api: Web interface,” Accessed: Jun. 19, 2024. [Online]. Available: https://typeapi.lab.pidconsortium.net/.