Automated Schema Extraction for PID Information Types

Ulrich Schwardmann
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG)
Göttingen, Germany
ulrich.schwardmann@gwdg.de

Abstract—Typing is a well known concept to prepare services for data processing for instance by choosing the correct service to a mime type for processing. But a lot more metadata elements, like availability and access conditions, provenance, processing preconditions or integrity parameters, are useful to be known in advance for preprocessing data services. In order to expose such metadata independently from data access, it can be stored directly within the persistent identifier structure pointing to the data itself. Such metadata is called PID information type. But the correctness of the type entries needs high accuracy, because all following services are relying on their expressiveness. Data Type Registries are emerging for describing such types, but currently there is no automated way to control the content of types. This work introduces an approach to automatically derive schemas from hierarchically organized type descriptions in a recursive way and highlights the advantages of the resulting precision in a couple of application examples.

Keywords-mime type; data type registry; persistent identifier; PID information type; JSON schema

I. INTRODUCTION

Identification and typing are two central concepts of data management. The reference to data objects by some kind of identifier is obviously needed to access data, and typing is used to prepare services to process the data, for instance by choosing the correct service for processing. Typical, and also in the big data context widely used examples of these concepts are URI/URL and mime types. But because access and processing are key functionalities for any kind of data management, the identification of data and its types are concepts with a much wider generality and possible impact than these elementary, but well known examples above.

The importance of identifiers grew during the recent years together with the emergence of different identifier systems for different purposes, like the unique identifiers inside object storage systems or the stable references to digital objects given by persistent identifier systems.

Types on the other hand are a kind of metadata for digital objects, called key metadata in [6], and a lot more of such metadata elements exist, that would be useful to be known in advance to preprocess data services, than just mime types. Typical such simple but useful instances are provenance or access right informations, checksums, time stamps and frames and geolocation coordinates used for different purposes, versions and provenance or processing preconditions like other data needed for interpretation. Such metadata used especially for data preprocessing are called information types and are helpful parameters in building complex workflows on data management, data transfer and data processing.

Mime types are distinguished by a well defined name extension (RFC2046) of the ressource. But to allow a much broader usage of types, as suggested above, this approach is not flexible enough. The type information has to be stored somewhere else, and because the information about the structure and conditions of the ressource is often necessary before processing or even transferring of the data, it should be close to the reference information. A consistent alternative for a broader use of types is therefore, to couple the type information strongly to the persistent identifier (PID) that points to the ressource.

Only standards defining the use of such types can lead to a high level of interoperability between the systems that use them. In the case of mime types the definitions can be found for instance as RFCs (in this case RFC2046, 6838, 4289 and 6657). Similarly for information types it is necessary to establish standards, to register such types in a data type registry and to control the content of type instances.

In working groups of the Research Data Alliance (RDA)\(^1\) a general framework for this kind of data typing, the registration of data types\(^2\)[7] and the concept of information types used together with PIDs\(^3\)[14] was proposed. A data type registry software (a prototype of the software used in this context), and an interface description for the access to information types were practical outcomes of these working groups and Data Type Registries describing such types are now emerging.

The information stored in type instances has to be particularly reliable, because the functionality of the data services is dependent on a correct preprocessing, which again relies on the expressiveness of the type. Therefore the correctness of the type entries needs high accuracy. But currently there is no automated way to control the content of types by schemas.

This work introduces an approach to derive schemas automatically from the type descriptions, which simplifies the

\(^1\)http://rd-alliance.org
\(^2\)https://rd-alliance.org/groups/data-type-registries-wg.html
\(^3\)https://rd-alliance.org/groups/pid-information-types-wg.html
schema definition significantly and leverages the accuracy of the chosen services. The advantages of the resulting precision of types and the use of precise types for data management and processing are highlighted in a couple of application examples. Conformance checks, as envisioned in [14], are easily done this way and can be seen as an additional component in the typing framework of RDA.

A. Schemas as Part of the Information Type Description

During preprocessing the type metadata is interpreted by automated services, and therefore it is necessary to avoid each precondition of human interpretation. It has to be defined in a clearly determined and reproducible way. Machine driven processes have to rely on the correctness of the given metadata to handle the data.

To provide syntactical correctness proofs for information types by schemas is feasible and enables the preprocessability for the services. The existence and correctness of schemas, proving the syntax, is therefore crucial for the applicability and reliability of such preprocesses and the acceptance of information types.

This implies, that for the operation of an information type registry it is not sufficient to register the types itself. A framework to easily construct and store the schema definitions in a reliable and openly accessible way is also necessary. A natural place to provide the schema of a type is its type definition itself, because this has to be reliable anyway, is easy accessible and all information about the type should be available there.

The construction process of schemas as described below can be seen as language independent on a conceptual level. But because the Handle® System uses JSON as representation layer for its object identifiers, it is a natural choice to use also JSON Schema in this context. An XML like schema construction however would require an adapted set of keywords in the type description in order to provide the XML schema possibilities.

1) Schema Construction: Only a flexible and transparent type definition user interface and an automated process, constructing schemas directly from the information type definition, is able to provide the necessary type framework for information types, that can be expected to have a wide variety emerging from many community specific data management workflows.

The possible valid instances of information types can be diverse, as one can see already for instance in the simple case of checksums with MD5, SHA and others. The often found approach, to derive schemas from instances, is therefore not of particular help here, because it tends to incorrect results by overfitting to special instances. Because it is not an instance of the type, also the type definition itself is not be used with this approach of course. However it can be the correct source for schema derivation, if it is provided with all the information needed for such an automated process.

![Figure 1: Subtype hierarchie of the geographic-coordinate type](image)

Information types are often referring to simpler types. For instance geolocations contain longitude, latitude and altitude and the former can be given in sexagesimal or (semi-) decimal form (see figure 1). This can be seen as a generic pattern of metadata dependencies. Therefore a hierarchical structure of the type definitions, where also the referred subtypes already bear the information they need for their schema, is suggested here to provide all the information needed to derive the schema.

Such an approach however needs well defined and clearly structured descriptions, how a derived type is exactly built out of subtypes, and how to retrieve all necessary information from the subtypes. The construction of this framework for describing correctly these dependences, enabling all the desired properties of even rather complicated type definitions and its schemas, processing the type definitions to extract the schemas and then providing the schema as part of the information type definition, is described in the following.

B. The Technical Background of the Implementation

The descriptions of types need so called data type registries as stable and transparent infrastructure and must be reliably referable. Consequently they need to have a PID as well.

One way to directly couple type information of the data to its PID reference is to utilize the database behind the PID systems, and to specify there additional entries about useful, general properties of the digital objects. This approach can be realized for example with the Handle® system,
where arbitrary information can be stored in the data base together with the persistent identifier.

For such PID information types a first implementations of a data type registry is now available\(^4\) and operated by the ePIC Persistent Identifier Consortium for eResearch\(^5\). A couple of prototypical types are already hosted there.

The implementation is based on the Data Type Registry software, distributed by the Cordra project\(^6\), hosted at the Corporation for National Research Initiatives (CNRI)\(^7\). It was designed in a design study already in [8] as a digital object management software that provides techniques for managing digital information. A prototype was an outcome of the RDA Data Type Registry (DTR) Working Group [7]. The Cordra software can be configured to accept metadata records that conform to types and provides a RESTful API as described in the Cordra Technical Manual[3]. Accordingly configured a Cordra instance provides a PID to each type by using again a Handle\(^\text{®}\) server backend. The information type description itself is coded in JSON.

The extension of the existing DTR concept of type definitions in Cordra to allow an automated extraction of schemas for each individual type definition is made backward compatible, to enable federations with other type registries. The ePIC implementation uses JSON schema as a powerful tool for describing and validating the structure of data and embeds it into the JSON object of the type definition.

II. Hierarchical Type Definition

As described above a precondition for the automated schema extraction suggested here is a hierarchical structure of the definitions allowing information types to be derived from well defined other information types.

A. Hierarchical Metadata Descriptions

Such a hierarchical approach for defining metadata is not new, there exist already some approaches for building hierarchical metadata schemas. One prominent example is the Learning Object Metadata (LOM)\(^8\)[1], that comprises a hierarchy of elements, that may be simple elements that hold data, or may themselves be aggregate elements, which contain further sub-elements.

Another example is the Utah Metadata Application Profile (UMAP)\(^9\), that refers to LOM, has a detailed map[10] of the hierarchy of the metadata used and produces a general description[11] of the hierarchical relationships and element interdependencies they use.

A third example is the Component Metadata Infrastructure (CMDI)\(^12\)[2] of the CLARIN project\(^10\)[12], where CMDI addresses the needed flexibility by using metadata components with agreed and registered categories, based on the ISOcat data category\(^13\), where a component can also contain one or more other components.

And there are other examples like Crossref\(^13\) [9] or DataCite\(^14\)[5], which actually define their metadata schema in a hierarchical way without noting it explicitly. Even in Dublin Core\(^15\)[13] there are hints of a hierarchical structure given by the “Element Refinements”, as refined elements share the meaning in a more restricted way.

In these approaches above a schema definition is made for some metadata with a given name. Used as a substructure it is referenced in some other metadata by this given name. For the validation of instances such a reference itself is not sufficient. It is in fact necessary to completely include the sub-metadata schema into the new metadata schema. In some such cases new additional and previously undefined keywords are introduced in order to fulfill the actual requirements in the metadata description. An example is the formerly undefined keyword “titles” in the DataCite XML metadata schema [4] in order to reflect the (1 − n) requirement for the “title” in the schema description[5]. Such an enhancement needs semantical knowledge and is probably not performed automatically.

Beside such syntactical gaps between schema description and schema definition as above an additional difficulty to derive a schema from subschemas is often, that no reliable and standardized way is available, how to refer to subschemas. This can be changed in particular with the use of PIDs for types, as we will see below in the description of the necessary framework.

In all these approaches above the schemas are mainly constructed “by hand”, where the hierarchical structure is of great help of course. To our knowledge an approach for schema extractions, where the hierarchical structure is used by an algorithm to provide the schema, does not exist until now.

B. Metadata Schemas for Information Types

A sufficiently rich description providing all the information about the type itself, its referred types and the relationship between them is required for an automated schema construction out of formerly given types. In the following we provide some formal definitions needed for this framework.

\(^{4}\)http://dtr.pidconsortium.eu/

\(^{5}\)http://pidconsortium.eu/

\(^{6}\)https://www.cordra.org/

\(^{7}\)http://www.cnri.reston.va.us/

\(^{8}\)http://ltsc.ieee.org/wg12/

\(^{9}\)http://www.uen.org/dms/umap/

\(^{10}\)https://www.clarin.eu/content/component-metadata/

\(^{11}\)http://www.clarin.eu/

\(^{12}\)http://www.isocat.org/rest/dc/2597

\(^{13}\)http://www.crossref.org/

\(^{14}\)https://www.datacite.org/

\(^{15}\)http://dublincore.org/schemas/xmls/
1) Types: The phrase type plays an important role here and we will use it in different meanings in the following.

Types can be any possible kind of metadata for digital objects. Subtypes (and subsubtypes) are types referred by other types in a hierarchical type structure.

Information-Types are types of digital objects used to preprocess the data to prepare transfer and/or processing of the data.

PID information types (PITs) are information types, which always have a persistent identifier assigned to it, and where the instances of the PITs are stored together with the persistent identifier of the typed digital object.

With JSON types we mean only those special types allowed inside JSON and defined by the JSON standard, like boolean, number etc. .

If the meaning is clear and the context allows such a generalisation, the phrases types and subtypes are also sometimes used for more specific types like PITs in the following.

2) Hierarchical PID information types: A PIT is built out of a finite combination of PITs and Basic PITs, which again do not rely on other PITs and only use the elementary JSON types (no objects and arrays) with possible restrictions given for example by regular expressions.

The Basic PITs are called properties in [14], but types are not explicitly described as a recursive concept there.

The combination of types is given in a structured way according to the JSON schema keywords and rules. Each non Basic PIT only describes the dependencies of the used next level of PITs in its registered description. The structure of a PIT as a derived type is given by dependencies to its subtypes and that of the subtypes to their subsubtypes. The kind of dependency is given in the type description of each PIT, registered in the Data Type Registry. Dependent PITs are named and referenced by the PID assigned there to these subtypes.

3) Informal Type Descriptions: In the registered description of a PIT some metadata is more informal, like the description or information about the provenance of the PIT description. The list of related standards and its nature of applicability is also seen here as informal, because the most standards are defined in such a way, that an automated schema extraction is difficult if not impossible.

A bit less informal, but also not used in the schema derivation is the unit description and its dimension, if used, because this can and frequently will be used as an additional parameter, when exporting the metadata to other data environments. Some others, like identifier or the lastModificationDate for instance are updated or assigned automatically.

But there are several metadata elements in the type description that are directly related to describe the dependency structure, some already given in the Cordra implementation, some as an extension.

C. References, Names and Keys

Because each subordinated PIT is always an explicitly defined and registered object with a PID and an internal substructure, each reference can be given explicitly by its PID.

PITs, including Basic PITs, additionally have a name in the property description. Also for the next level of substructures names are given in the property descriptions of the PIT description. These names become the keys in object instances defined by the schema for the given PIT and are of a special relevance for the schema therefore.

Where the structure of the instances of PITs is determined by keys, the allowed values of the key value pairs of the instances are again defined in the schema by the referenced PITs inside the subtype. These subtypes may have again a substructure for the instances and so on, ending eventually with the Basic PITs or recursive definitions as in figure 2.

The names for subtypes can differ from the names given as keys of the dependent types in the properties of the type, which gives the flexibility to reuse given PITs for new purposes. For instance a formerly defined type with name "unicode string" can be used in a derived type as comment field key refering by PID to the unicode string type, without referring to its name as a key.

Name collisions are easily avoided in this context. There is an important difference between the names here and those used in XML schema with the necessity to group them in namespaces. In the case of PITs always the reference, the PID, and not the name determines the type. Other PITs and schemas can use the same key for a subtype with a different meaning without a conflict, because the definition of that subtype inside the schema does not rely on the key but on the PID of the subtype, as we will see below. The type description and the schema belongs exactly to that type and keys in a schema do have the given meaning only inside that schema, such that each PIT has its own intrinsic namespace.

Whereas the previously described general method is language independent, the following concrete description of the keywords used to refer to corresponding schema constructions highly depend on JSON schema.

D. The description of Basic PID Information Types

Essentially the JSON-Types and restrictions, which are allowed by JSON schema, are available as allowed values of a Basic PIT, because they don’t refer to other PITs. These are the elementary JSON-Types "boolean", "string", the numeric types "integer" and "number". Also a Basic PIT can have a value out of an "enum" list of possible constants, not necessarily all of the same elementary JSON-Type.

Restrictions like multipleOf, minimum, maximum, exclusiveMinimum and exclusiveMaximum can also be defined for the numeric elementary JSON-Types.

For the elementary JSON-Type "string" there are restrictions possible by regular expressions (ecma-262) and
additionally by the minLength and maxLength validation keywords.

In the JSON object of a registered Basic PIT description the keys “dataTy pe”, “enum”, “restrict”, and “regexp” are used to describe these possible values and restrictions and make them usable for the automatic schema extraction.

Arrays and objects are not available for Basic PITs. These two structures of JSON types are major building blocks for PITs.

E. The description of PID Information Types

A JSON object consists of an arbitrary number of key value pairs and a JSON array consists of an arbitrary number of values. Both JSON types are structured by according separators, and recursively each value again can be any JSON type. PITs are in principle organized in the same recursive way, as they consist of a number of PITs structured according to the definition and semantics of JSON schema.

The structure itself is represented by dependencies of given PITs to formerly defined PITs. A sufficiently expressive framework is needed to reflect the different allowed dependencies. Two sections in the registered type description are used for this: one with the JSON object key “representationsAndSemantics” and one with the key “properties”.

1) Representations and Semantics: In the “representationsAndSemantics” part we define by “subSchemaRelation”, how the elements given in the “properties” part are related to each other, for instance as array, as objects with property structure or with choosable single properties. For an object structure additional properties can be allowed or not and restrictions on the property selection can be expressed, like all, any or exactly one of the properties must be valid for an instance.

This is done in the type description with keys similar to the well defined control keywords of JSON schema, like denyAdditionalProperties to describe objects or isArrayWithGivenProperties to define arrays.

a) Defining Arrays: If only one property is given, the array contains arbitrary many elements of only the given property. In the JSON schema description this is called list validation. The number of elements in the array can be restricted with uniqueItems, maxItems or minItems.

If more than one property is given, the array contains a tuple of the given properties, which is called tuple validation in contrast.

With list validation and maxItems = 1 a one tuple array can be defined. A combination of both, a tuple of the given properties together with an array of one property, or in other words a combined list and tuple validation is not allowed, neither in JSON schema nor in the registered type description structure here. Such a combination has to and can be layered by accordingly given subtypes.

b) Defining Objects: To define an object structure given by properties, the type description keys requestAnyOfProperties, requestAllOfProperties, requestOneOfProperties or denyAdditionalProperties are used.

The number of subtypes required in the type object can be restricted with minProperties and maxProperties. Additionally the subtypes itself can have the obligation to be mandatory or not.

Restrictions on the property selection, whether all, any or exactly one of the subtypes must be valid for an instance, are expressed with the requestAnyOfProperties etc. keys in the type description. Also the isNot key in the registered type description expresses the restriction, that an instance validates against neither of the given subtypes.

Because of possible contradictions the additionally possible obligations of the subtypes, like “mandatory” (s.b.) are ignored in all these latter cases.

2) Properties of the Subtypes: In the “properties” section however the subtypes are listed as elements together with additional element dependent parameters like their obligation, where each of the elements is assumed to be either mandatory or optional. The elements are named to define the key in the resulting schema and they are represented by references (PIDs) to formerly defined PITs or Basic PITs.

There is also again a subsection “representationsAndSemantics” defined inside each “properties” section, where one choses, whether the subtype is “repeatable” and the “obligation” defines whether this subtype is mandatory or not. The elements “expression” and “value” have the more informal character, to declare the purpose, this property is used for and its dimension. They are not used for the schema extraction, but for interoperability with other data type registries (see also instances in PID-InfoType-schema.json).

F. Additional Features

1) Allow Abbreviated Notation: Data is often not given as an object with key value pairs, but strings as values to the PIT as key or by arrays or lists of the values with an implicit assignment of the value to the key by the order in the list. Therefore for derived PITs it might be useful to allow such an abbreviated notation. This only makes sense for PITs defined as objects, where additional properties are not allowed and where all properties have to be given, because otherwise the implicit assignment of the value to the key would not be possible. For these cases one additionally defines such an array structure in the schema consisting of all registered subtype descriptions in the order of its description, and allows both the array and the property description in the schema of the type.

2) Unnecessary Property Levels and Subsidiary Structures: By the hierarchical approach chosen obviously properties and items on each level always have to be given by an already known PIT or Basic PIT in a key value pair,

http://dtr.pidconsortium.eu:8081/objects/21.T11148/b72cf35b541e2ef79830
which leads to repetitions of names in the keys. In principle this is only a syntactic and not a semantic restriction of this approach, because it might only introduce additional levels of nesting and keys. But it can be the case that there are intermediate keys involved, that seem to be unnecessary, such that one might want to avoid them.

As an example a given generic substructure \textit{struct} like \textit{‘printable-string’} should be used for example in a structure \textit{struct1} with a specific name like \textit{"title"}, to which another \textit{struct2}, like \textit{“bibliographic-metadata”}, refers again with the name \textit{“title”}. Such chains of structures with the same name, that are used as keys of the instances, are often useful to clarify the dependencies of schema definitions, but are unwanted as chains of same keys in the explicit instance that fulfills the schema definition. In such a case one needs a way to optionally omit names and refer directly to substructures.

In another example one could want to define a structure \textit{struct0}, consisting of a choice of different substructures \textit{struct1}, \ldots \textit{structk}, and give it a generic name. To distinguish the substructures it is necessary to name them in the registered description of the structure. But for the use of \textit{struct0} these names are unnecessary and in general unwanted, because the name of \textit{struct0} should be generic for all the possible choices, which means that the substructures should not be named explicitly. In the geographic-location PIT this is the case for example. Again one needs a way to optionally omit these names and refer directly to substructures.

This option can be indicated in two ways, by using exactly the same name again in the reference or by using the data type keyword \textit{“allowOmitSubsidiaries”}. But a prerequisite for this of course is, that the name can actually be omitted without losing structure, which is only the case, if the child has exactly one and not a list of property sections, or a choice out of such a list is required. In terms of directed graphs it is only allowed to omit such a node, if the node has just one child and is superfluous there in a topological sense.

III. AUTOMATION OF THE SCHEMA CONSTRUCTION
A. The Schema Extraction Algorithm

The algorithm retrieves the PIT description and uses the keys, described above, to extract the schema. Its overall structure is a simple backtracking algorithm, because the construction of a PIT is recursive as described above with a PIT built out of a combination of PIT and Basic PIT.

On start of the algorithms, invoked with the PID of the main type, the PIT description is retrieved from the PIT registry and validated. The properties section with references into the definitions section generated from the description and an empty list of definitions is initiated. This list is expanded during the schema construction and contains the definitions of the main type, all subtypes and subtypes of subtypes.

The recursive backtracking function starts with the current PIT and is called by the PID as the reference to a PIT. In the preprocessing part of this function again the PIT description is retrieved from the PIT registry and validated. The entry in the definition list is generated for the current PIT according to the type definition keys found and with the PID as reference value of the definition.

The subtypes in this definition are referenced by their PIDs. After preprocessing the backtracking function is called again with each PID of the subtypes found in the registered type description, which establishes their definition again in the definition list. If a PIT does not contain a subtype, it must be a Basic PIT, which also gets by preprocessing an entry in the definitions list, but where the recursive structure obviously terminates without further function call.

After all recursive calls are finished, the schema of the main PIT is built containing a JSON definitions section with definitions of all subtypes and subtypes of subtypes referenced by their PIDs and a properties section containing the structure of the main PIT as given in the first step.

An implementation of the extraction algorithm is written in Python and comprises about seven hundred lines of code.

B. Termination and Infinite Recursion

Registered PIT descriptions can contain subtypes of subtypes that itself call higher level subtypes, which leads to cycles in the hierarchy graph.

This actually can be a desired structure for a PIT description, but it would lead to an infinite recursion in the version
of the algorithm as it is described above. One can avoid this
by simply omitting the recursive call to a subtype, if this
subtype has already an entry in the definitions. In this case
the recursion depth is trivially finite if one makes the not
unrealistic assumption, that only a finite number of PIDs is
actually registered, but it also can be proven inductively for
infinite PITS as below.

**Proposition 1.** With a finite number of basic PITS the num-
ber of subtype and subsubtype PIDs, called by recursion for
a given hierarchical PIT, is finite for the schema extraction
algorithm above.

**Proof:** A PIT together with its subtypes and subsub-
types is abstractly a directed graph with types and subtypes
as nodes and a subtype relation as edge.

For each PIT we can build the spanning tree of its
subtypes, which contains all subtypes, which do not have
references to subsubtypes of a lower recursion depth, than
the recursion depth of the subtype itself.

Now the proof is given by induction. For basic PITS the
claim is obviously true, which gives the induction start.

For the induction step assume that the claim is true for
all PITS with a spanning tree of subtypes upto a recursion
depth of \( n \) and assume, that this is not the maximal recursion
depth that can be found for PITS. Otherwise in the induction
step the same argument can be used for \( n + 1 \) to \( n \) or in the
case of \( n = 1 \) the claim is hold by finiteness of the basic
PITS.

For all PITS with a spanning tree of recursion depth
of \( n + 1 \) (which exist by the assumption that \( n \) is not
the maximal recursion depth) we know that the number of
dependent subtypes is finite by definition (see subsection
II-B2).

Therefore the maximal number of subtype and subsubtype
PIDs of this PIT is maximally the sum of the subtype PIDs
and the finite sum of the finite number, that each subtype
brings in, which is in total a finite number.

Because the number of PIT subtypes is finite by definition
(see section II-B2) and with a spanning tree argument in the
directed graph of the PIT with types and subtypes as nodes
and a subtype relation as edge.

**Proposition 2.** With a finite number of basic PITS the
algorithm as above, omitting recursion at already visited
subtype PIDs, terminates with a complete schema definition.

**Proof:** If a subtype has already a reference in the defi-
ition list, also the backtracking function was called before
for the subtypes of that subtype. Thus by the recursiveness
of backtracking also all subsubtypes have already reference
entries in the definition list. Therefore for a PID, that is
already in the definition list, the backtracking function needs
not to be called again in order to complete the definition list
for the schema.

Since the algorithm starts at a given PIT and since it does
not call the backtracking function for a PID, that is already
in the definition list, lemma 1 shows, that it terminates after
the finite total number of subtype and subsubtype PIDs in the
definition list with all necessary references for the schema.

However, even if this algorithm operates only a finite
number of steps, and the definition list in the resulting
JSON schema contains only a finite number of references,
the registered descriptions still allow instances with arbitrary
recursion depth as desired, because the recursive structure
is still given by the recursive reference in the definition list.
The “recursion-example”\(^\text{17}\) (see Figure 2) is a minimalistic
example of such a recursive PIT structure.

C. Testing

During development and adaption of the Schema Ex-
traction Algorithm it was and is necessary to check code
changement directly with the outcome of the schemas
against test instances. For the possible code changements
two cases of impact on the type schemas this might have:
for some types there must be no change on the outcoming
schemas and others are expected to be changed by them.

Therefore there exist two testing procedures, one that
checks for all existing schemas, whether they are changed
by the code changement, and another that checks whether a
schema is valid for a set of correct examples and invalid for
another set of false examples. Both test procedures use the
unitest framework. The valid and invalid instances for all
given schemas can be found\(^\text{18}\) at the PIT registry of ePIC.

IV. Examples

Figure 2 shows a schema for a simple PIT that has only
one subtype, which is in this case the type itself because it
is also used above as an example that recursion is possible.
Here also a couple of correct instances for this type is shown.

Figure 1 a and b shows the dependencies of the PIT
graphic-coordinate\(^\text{19}\). The coordinate consists as usual
of latitude, longitude and altitude, where the altitude is a
Basic PIT. Latitude and longitude can have a decimal, a sex-
agesimal and a semidecimal (degree with decimal minutes)
representation. Each such representation has a Basic PIT
with accordingly restricted regular expression and the PITS
for latitude and longitude allow any of these representations.

The corresponding schema definition one can retrieve also
directly from the type registry.\(^\text{20}\)

\(^{17}\) see http://dtr.pidconsortium.eu:8081/objects/
\(^{18}\) http://dtr.pidconsortium.eu/samp/
\(^{19}\) see http://dtr.pidconsortium.eu/xmpls/
\(^{20}\) http://dtr.pidconsortium.eu:8081/objects/
\(^{21}\) T11148/5105f4d67ba0b5c5d08
The previous-version type 21 allows to visit previous versions referenced by an identifier, also iteratively if this type is present in the referred previous versions.

Another more complex example22 is the JSON equivalent of the mandatory metadata elements of DataCite, where an easy metadata crosswalk can map instances between these different metadata descriptions. This kind of type is used for interoperability between ePIC and DataCite metadata.

A final example shows the benefit of this typing concept for the users in validation of syntactical correctness for tabular data, a wide spread data format, as shown in figure 3. The types of the columns are given as argument by a PID list of types.

Used here is a CSV file23 for locations of capitals, which was slightly modified to show the potential of elaborated regular expressions by interchanging the latitude with the longitude, leading to a syntax error for the latitude. The other type errors are actually found in the original file and are due to missing country codes and in the Washington case due to an appending D.C. as usual with a comma separator in a CSV file.

This example shows how PITs can be used for data preprocessing in this case to give a first control about the accuracy and applicability of the data for further processing.

V. LIMITATIONS

This implementation of schema extraction for such hierarchical structured PITs has of course some limitations. Some are dependent on the current state of the implementation and some are of a more systematic character.

A. Limitations with Respect to the Input Validation

Currently there is no control and validation or plausibility check of the registered type description at the API or Type registry level. As a consequence an inconsistent input can be defined. This does not lead to inconsistent schemas, because the extraction algorithm omits, if possible, contradictory type description keys like subtype obligations, otherwise it gives out an error. But this is part of the algorithm and may possibly lead to unexpected results in the schema definition. Therefore a tighter integration of the possible and allowed combinations of the structural elements in the registered type description front end would be helpful and desirable in a next version of the software.

B. Limitations with Respect to the JSON Schema Standard

Some advanced features of JSON schema are not implemented yet. Property dependencies, which declare, that certain other properties must be present, if a given property is there, and schema dependencies, which declare that the schema changes when a given property is present, are not available as key value pair option in this version of the implementation.

These features could be added later if necessary and such an extension could be made upward compatible, where it would only be used in new PITs and would not effect the structure of existing PITs. But currently such constructions seem to have only a lower impact. It seems that just with the hierarchical approach one often can construct an equivalent structure of dependent types and subtypes by defining subtypes.

Also pattern properties are left out in this version of the schema extraction for PITs. Because extra properties lead to a higher degree of undefined possible syntactical constructions and allowed subtypes need to be specified, the possibility to restrict the names of the extra properties seems not appropriate for PIT descriptions.

Array resp. objects are currently always defined with additionalItems : false resp. additionalProperties : false, which is not seen as a mayor drawback, because allowing additional unspecified items or properties means less control and PITs are assumed to deliver a high accuracy for syntactical correctness. Such options would lead to undefined possible syntactical constructions. It is of course possible to allow such uncertainty in later versions, if it is demanded.

VI. CONCLUSION AND OUTLOOK

The PIT Registry at drt.pidconsortium.eu realizes a hierarchical approach for defining a special class of metadata of digital objects used for typing. This approach is organized to allow the automated extraction of schemas for the types,
that can be used directly to validate instances of the type. Because it is done step by step and the options are well arranged at each level, the often cumbersome process to define appropriate schemas of types is easily and much more reliable given here by an automated process. In principle this approach is not limited to JSON or to PITs, but here it is of particular importance, because PITs are used for preprocessing of data and need to have a high reliability in the correctness of the information.

The work described here shows that automated schema extraction is possible, that an algorithm is implementable and is a finite process for arbitrary PITs with a finite number of subtypes. It further shows, as seen in the examples, that with this approach schemas for elaborated type metadata can be created with a lot of flexibility. A deeper integration of this schema component into the PID API will additionally improve the accuracy of PITs. As seen in the section about limitations there is some need for further code development and integration.

In all earlier approaches that rely on a hierarchical metadata definition, as described in section II-A, the schemas are constructed “by hand”. To the knowledge of the author the approach described here to systematically exploit the description of a metadata type to automatically generate a schema for that type was not given before. It is currently implemented at the ePIC PIT Registry.24

The examples show, that this approach has a great potential in the context of big data to extend interoperability between different domains of data referencing, to clarify access possibility and resource types, to proof data in advance on integrity, correctness and consistency, to easily explore earlier or later versions and the provenance, to describe data and service dependences and to prepare services for later processing of data.

The ePIC PIT Registry can be used by data managers to define PITs. Current types are defined as candidates in the Candidates PIT Registry and are still due to possible changes during its revision phase. After approval they get a new PID and their content gets fixed.

ACKNOWLEDGEMENT

The author would like to thank Tibor Kálmán, Robert R. Tupelo-Schneck and Giridhar Manepalli for helping to setup the PIT registry, and also together with Larry Lannom and Daan Broeder from the RDA Data Type Registry working group and Tobias Weigel and Thomas Zastrow from the RDA PITs working group for all the fruitful discussions. Furthermore thanks to GWDG and to ePIC for providing the infrastructure for the PIT registry and last but not least to Ulla always encouraging me with great patience.

REFERENCES


24see: http://dtr.pidconsortium.eu