# ACTIONABLE PERSISTENT IDENTIFIER COLLECTIONS

***Tobias Weigel*[1,2*]*, Stephan Kindermann*[1]*, Michael Lautenschlager*[1,3]

[1]*Deutsches Klimarechenzentrum, Bundesstrasse 45a, 20146 Hamburg, Germany*
**Email:* weigel@dkrz.de
[2]*Universität Hamburg, Bundesstrasse 45a, 20146 Hamburg, Germany*
[3]*World Data Center for Climate, Bundesstrasse 45a, 20146 Hamburg, Germany*

## ABSTRACT

*Persistent Identifiers (PIDs) have lately received a lot of attention from scientific infrastructure projects and communities that aim to employ them for management of massive amounts of research data and metadata objects. Such usage scenarios, however, require additional facilities to enable automated data management with PIDs. In this article, we present a conceptual framework that is based on the idea of using common abstract data types (ADTs) in combination with PIDs. This provides a well-defined interface layer that abstracts from both underlying PID systems and higher-level applications. Our practical implementation is based on the Handle System, yet the fundamental concept of PID-based ADTs is transferable to other infrastructures, and it is well suited to achieve interoperability between them.*

**Keywords:** Data-intensive science, Persistent identifiers, Unique identifiers, e-Science infrastructures, Scientific data management

## 1 INTRODUCTION

Colloquially, a Persistent Identifier (PID) is a globally resolvable unique name associated with a digital object. *Persistency* here means that if the object relocates to a different server or owner, the identifier name remains the same. PIDs have traditionally been a topic of interest in library science and the scientific publishing business such as, for example, within DataCite (http://www.datacite.org). More recently, the demands of data-intensive science (Hey, Tansley, & Tolle, 2009) have driven several scientific disciplines to the idea of employing PIDs for scientific data management. Large infrastructure projects, such as EUDAT (European Data Infrastructure, http://www.eudat.eu) or DataONE (Data Observation Network for Earth, http://www.dataone.org), as well as domain-specific projects and community activities, such as DCO-DS (Deep Carbon Observatory Data Science, http://tw.rpi.edu/web/project/DCO-DS) or CLARIN (Common Language Resources and Technology Infrastructure, http://www.clarin.eu), have been assigning identifiers to the first-class objects they are dealing with or plan to do so in the near future. Some of the first working groups of the RDA (Research Data Alliance, http://rd-alliance.org) focus on exploiting PIDs and building technical solutions around them.

Within these activities, PIDs are no longer assigned only at the late data life cycle stages of long-term archival and preservation but increasingly at the intermediate stages of dissemination, analysis, and synthesis of research data. This results in a variety of new usage scenarios that transcend the original use cases of PIDs. PIDs are generated in much higher numbers and at higher frequencies, and applications are shifting towards core data management tasks throughout the data life cycle. Therefore, the overall demand for PIDs that can be managed and are actionable beyond a mere lookup operation is increasing. Moreover, the focus shifts to machine actors instead of human users who need to access and organize PIDs and the objects behind them. Automated systems, such as the integrated Rule-Oriented Data System (iRODS, http://www.irods.org) described by Rajasekar, Moore, Hou, Lee, Marciano, de Torcy, et al. (2010), are increasingly seen by data centers as promising solutions to deal with the data deluge.

In this article, we aim to provide a conceptual framework and practical implementation for managing large amounts of PIDs. The key idea is to combine the widespread assignment of PIDs with well-known abstract data types (ADTs) that aggregate the formerly isolated PIDs, much in the way described by Kahn and Wilensky (2006), as objects with data type *set-of-handles*. We will extend this concept and show how a multitude of collection-like ADTs can be implemented based on the Handle System (http://www.handle.net) data model. Aside from the exemplary implementation, the concept is not bound to a particular PID infrastructure but, moreover, should be seen as a means to provide interoperability between different PID infrastructures. We strongly encourage a view where ADT instances with operations that work with PIDs should become the pivot point for interoperability across such infrastructures.

The article is structured as follows. We will provide more details on the well-established concepts of ADTs and PIDs in Section 2. Section 3 presents a set of detailed use cases that we will analyze in Section 4 to come to a core conceptual framework, including a small number of crucial requirements. In Section 5, we will present our practical implementation based on Handle System records, which conforms to the conceptual framework. We will compare our findings with related efforts in Section 6 and provide some final remarks in Section 7.

## 2 FUNDAMENTAL CONCEPTS

In this section, we will provide an overview of the two well-established concepts fundamental to the core of this article.

### 2.1 Abstract data types

Abstract data types (Liskov & Zilles, 1974) have been a concept fundamental to modern software libraries and the archetypical programmer's toolbox over the past several decades. Commonly known abstract data types are, for example, Lists, Sets, Maps, Trees, and Graphs[1]. The key characteristic of ADTs, the method of abstraction, is that they are not defined by their storage representation but solely by the operations they offer – they are actionable by their very definition. In the end, the actual storage structure is determined by choosing a particular implementation model. For each ADT, alternative implementations, such as arrays and linked lists for the list ADT or binary search trees and hash tables for the set ADT, may and should exist. The actual decision of which implementation to choose for which ADT often requires trade-offs regarding operation efficiency while most importantly the core functionality of an ADT, defined by its operations, remains unaffected by the underlying implementation.

For this article, we consider five important and very common ADTs: Lists, Sets, Maps, Trees, and Graphs. We will just quickly recapitulate those of their properties that are most relevant for this article; more thorough descriptions can be found in common introductory books on data structures.

Sets are unordered and may not contain duplicates, as opposed to Lists, which are always ordered but can contain multiples. Maps consist of *(key, value)* pairs so that each key appears at most once. A Tree consists of nodes, where each node bears a payload value and relates to a number of unordered non-duplicate child nodes. A node with zero related child nodes is a leaf; every Tree has a root node. A Graph consists of a set of nodes and edges. Every node bears a payload value and is related to a number of edges. Every edge is related to exactly two nodes and is defined as pointing from the first node to the second node. Every edge can also bear a symbolic label.

The operations supported by these ADTs are often similar, which is an advantage when trying to promote their usage as a common interface layer. All of them offer some forms of add, insert, update, and delete operations, and in particular Lists and Sets are very similar in this respect. Differences lie in their efficiency.

### 2.2 Persistent identifiers

To explain what characterizes Persistent Identifiers, we first need to look at the broader concept of Digital Objects. A Digital Object as described by Kahn and Wilensky (2006) is from a technical viewpoint an entity whose parts are *digital material*, *key-metadata*, and a mandatory *handle* as part of the key-metadata. The handle acts as the identifier of the object. Digital Objects can be seen as being at the heart of the preservation mission that, for example, digital libraries and long-term archives are pursuing. We call the digital material a *resource* if it is referenced through a URI (cf. the definitions provided by the World Wide Web Consortium, http://www.w3.org/TR/2004/REC-webarch-20041215).

The handle is a persistent, unique, resolvable, and interoperable identifier (Paskin, 2009). The remaining key-metadata roughly encompass additional information associated with the data, and their entries have subsequently been called attributes, properties, or assertions. PID and key-metadata together form a Persistent Entity (PE)

[1] To avoid confusion with other concepts, we will refer to such exemplary ADTs by capitalizing them.

(Weigel, Lautenschlager, Toussaint, & Kindermann, 2013), which may be valuable enough to be preserved on its own terms even if the original digital material is gone. A Persistent Entity supports two fundamental conceptual resolution operations: locating the resource (at a single location) and retrieving additional information. Therefore, it is also an abstract data type in its own right and is a means to provide access to the additional information.

We will use the term PID for the identifier name, the term *Persistent Entity* for an instance of the ADT that supports PID resolution to data and associated information, and the term *Digital Object* for the combined entity of identified data, associated information, and identifier name.

Regarding identifier names, Paskin (2003) gives a good overview on the topic of the uniqueness of identifiers, questions on the required level of granularity, and in particular the question of identifier name semantics as was also discussed in the scope of URNs (Sollins & Masinter, 1994). Paskin defines such “intelligent identifiers” where the actual identifier string contains meta-information about the entity identified and finally largely advises against them. Our proposed solution should consequently not rely on the identifier name but should treat it as opaque, such as the names based on UUIDs (Universally Unique Identifiers, cf. ISO/IEC 9834-8:2008) used, for example, by DCO-DS and EUDAT.

## 3 USE CASES

In the following, we will describe some use cases that are taken from the Earth Sciences but apply to other domains as well. The reader should be aware that many peripheral issues mentioned in these scenarios are unanswered today, and it is beyond the scope of this article to discuss them in adequate detail. The focus is on how different scenarios can employ PID-based ADTs, and the use cases should be understood in view of this common theme. We also describe for each use case which ADTs may be used because later we must select a small set of ADTs to implement in our exemplary solution.

All use cases assume that all entities we are dealing with are Digital Objects that bear fully qualified PIDs.

### 3.1 Replication

Replication of research data is a task important for both long-term archival as well as data access and processing. Distributing copies of data to physically separate nodes located around the globe reduces the risk of data loss, enables more efficient access, and brings processing closer to the data. In this area, PIDs can help to identify all replicas of a master object. This scenario is one of the key use cases of the EUDAT project and is already actively put into practice based on PIDs.

Conceptually, the problem of replication can be described as follows. An original dataset at an originator's site must be replicated at $n$ partner sites, resulting in $n$ identical copies. To enable proper management of all replicas, such as synchronizing updates and taking care of lost replicas, some system must be designed to keep track of where the replicas are located and what the master object is. Obviously, PIDs can be used to store replica locations. Binding replicas and master objects together will, however, require additional information to be stored in close association with the identifiers. In EUDAT, an effective linked list implementation using Handle records is currently used to address this use case, but other ADT implementations are not offered yet.

Replicating data objects is one thing, but often they are also tightly associated with metadata objects, which may also be replicated. Even if they are not replicated, a central metadata record should be reachable from each individual replica. In practice, such things can become even more complex if data and metadata objects reside at different hierarchical levels, e.g., metadata describing aggregates of replicated data objects. Further below we will revisit these issues when we talk about combining different use cases.

The easiest way to bind replica locations together is through a List or a Set, depending on whether ordering is required. We also note that navigation should be possible from each replica object to the master object and vice versa.

### 3.2 Provenance of derived data

Data analysis in the Earth Sciences, and also in other disciplines, typically requires the use of processing tools, which transform or subset data, combine different sources, and so on. We select the following scenario:

Provenance information fragments are gathered at checkpoints where a data object progresses from one processing step to the next. At each such event, a PID is assigned to the data object and connected with the PIDs of predecessor data objects. Hierarchical grouping may be required if there is a multitude of distinct input or output objects.

The final result is an acyclic directed provenance graph of derivative data objects (Moreau, Ludäscher, Altintas, Barga, Bowers, Callahan, et al., 2008; Moreau, 2010), embedding once singular objects in a larger context. Since PIDs decouple from the storage location, this graph can transcend disciplinary and organizational boundaries. It also works well in mixed scenarios where some of the intermediate data objects are distributed beyond the original processing chain, e.g., through collaborative e-science infrastructures. Moreover, the graph continues to exist beyond the lifetime of often temporary intermediate objects if the corresponding PEs are preserved. This latter point cannot be emphasized enough in view of data-intensive science, where preservation of intermediate objects is unfeasible, yet provenance information becomes even more important.

In general, an important trade-off exists between granularity or detailedness of provenance information and the overall coverage[2]. Sophisticated schemas, such as the W3C PROV data model (Moreau & Missier, 2013), exist that allow for rich semantics and analysis, including context information beyond simple inputs and outputs, yet often the problem is gathering representable information in the first place. We state that a PID-based solution can achieve broader coverage, which is achieved at the intentional expense of detail. A significant amount of decrease in detail is desirable to ease cross-disciplinary and infrastructural adoption at this lower layer. More sophisticated schemas then form a higher layer that builds on top of the lower layer where gathered information is rich enough or additional sources are available. The initial problem for our scope here, however, is how to form graph nodes and edges in the first place and how to deal with hierarchical objects.

Regarding the question of which ADTs to employ, we provide the following argument. Describing the predecessor and successor objects of some pivot object can be done through a Map with controlled vocabulary terms used as keys, which also allows for storing additional context values. Alternatively, a Graph can be used to represent the whole provenance graph rather directly. Such a Graph must support edge labels to hold vocabulary terms. A List could be used as well, however, only in special cases where no element has more than one predecessor.

## 3.3 Composite objects

The large-scale use of PIDs is likely to result in situations where some objects that bear their own PIDs are grouped together to form a composite object that must also bear its own PID. One particular example for such a case is the hierarchy structure of the World Data Center for Climate (WDCC, http://www.wdc-climate.de) long-term archive. The WDCC hierarchy starts with high-level projects at the top and further subdivides into experiments and dataset groups down to individual datasets. Similar organization structures are used by community initiatives, such as CMIP5 (Climate Model Intercomparison Project, phase 5, http://pcmdi-cmip.llnl.gov/cmip5) and its distribution infrastructure ESGF (Earth System Grid Federation, http://esgf.org). In addition to a single hierarchy, scenarios exist where an individual object can be part of more than one composite object, e.g., a WDCC dataset can be part of a group as well as a separate experiment. When following ideas on how to employ PIDs to identify the different objects, assigning PIDs to all of these entities is just the first step. The second, more important step should then be to reflect the hierarchical structure by relating the PIDs to each other.

Regarding ADT usage, the first choice for the object hierarchy might be a Tree; however, in cases where an element is associated with more than one parent node, nested Sets or generic Graph nodes provide a more flexible solution.

## 3.4 Versioning

Today, much of the research data exhibit an increasingly dynamic life cycle. Data are acquired or generated and quickly disseminated through e-science infrastructures. For various reasons, however, an individual dataset may be recalculated and corrected at some later point in time with subsequent re-dissemination. In view of proper data management and documenting provenance, such data should be subjected to versioning: once a newer

[2] fraction of data documented vs. total amount of data

version of the same dataset is disseminated and further processed, the old version should not only be made obsolete but also cross-linked with the new version to document its history[3].

If old and new versions bear PIDs, the key question is what to do with the identifier assigned to the old version. One viewpoint is that the old identifier should be modified so it points to the new version; another suggests that a new identifier should be assigned. In reality, there may be a continuum, and choices depend on what the identity of a resource is and what amount of change is tolerable until the resource is seen as a new version. Here we will work with the scenario where a new PID is assigned. One important practical issue is then how we point from the old version to the new version and vice versa. The idea is to document the thematic link between the two objects and ultimately make it navigable.

In addition to separate identifiers for each version, users may want to always access the latest version of a dataset using a fixed identifier. Both scenarios are equally valid and can be required at the same time for the same set of resources.

To organize all versions of a Digital Object with ADTs, a List is an obvious choice because of the implicit temporal ordering and because we restricted the scenario so that there is no more than exactly one prior version. If this restriction is dropped, things get more complicated, and a Tree or nested Maps will be required.

## 3.5 Combining use cases

Before discussing the use cases, we will have a separate look at what happens when two or more use cases intersect.

### 3.5.1 Replicating composite objects and metadata

As mentioned earlier, replication often concerns data as well as associated metadata objects, both of which may be hierarchically structured. The hierarchies may differ, with metadata, for example, being defined with coarser granularity than subject data. This leads to situations in which metadata must be associated with composite data objects or cases where one hierarchy has more intermediate detail levels than the other one. Relationships may subsequently be defined in a more complex way than simply associating every metadata object with exactly one data object from an opposing level.

### 3.5.2 Combining archiving and versioning

A long-term archive's mission is to preserve the items given to it in their original state. As much as that is true, there is, however, also the question of what to do if data are re-submitted. Such cases happen all the time for various reasons, be it that digital-born data were re-calculated to fix errors only discovered after submission to the archive or because more data have become available extending an existing item (e.g., additional simulation variables from an Earth System Model). This is particularly true for e-science infrastructures used for data distribution in phases prior to long-term archival.

An archive is consequently obliged to apply versioning to the data. Distribution infrastructures such as ESGF can benefit from consistent versioning as well. However, given that archives and infrastructures contain composite objects, we get interference between these use cases when we end up having to version composite objects[4]. In the case of a long-term archive, the task is then to keep all original elements of the old composite available and also maintain the record that they belonged together as part of the composite, but at the same time the archive must also provide the new structure, with potentially new or even replaced elements. The new composite may turn out to be a combination of old and new elements (see Figure 1). An infrastructure that does not primarily address long-term archival, but still needs to keep information attached to obsolete PIDs (for example, provenance information), will face a similar problem.

[3] To keep things simple, we exclude more complex scenarios in which a dataset has more than one prior version (merge scenarios).
[4] The opposite case is possible just as well: building compositions of versioned objects

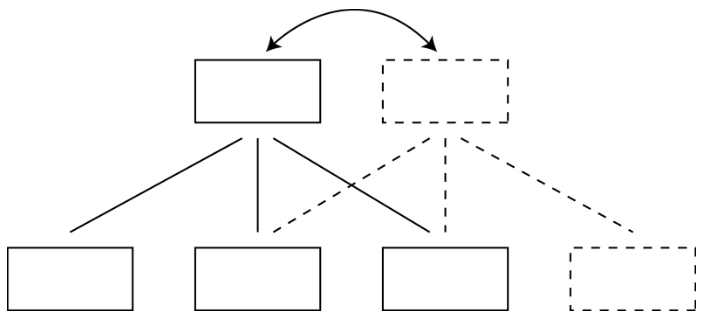

**Figure 1.** Versioning of composite objects. The solid object represents the first version, the dashed object the second version, where one object has been replaced with a new one. The composite objects need to be linked (arrow).

### 3.5.3 Combining processing and versioning

As explained earlier, provenance information about predecessor data whenever data is derived should be captured in at least minimal detail. However, in practice such processing steps are rarely done just once. Often we will encounter cases where a new version of a dataset is fed into the same processing chain, returning new results. Provenance documentation will then be incomplete unless we also bind the different versions of input and output files together. Doing so will establish two dimensions of provenance (Groth, Gil, Cheney, & Miles, 2012): the predecessor/successor dimension and the earlier/later version dimension. An individual dataset may be related to other elements along each dimension.

Moreover, such a dataset may even be a composite, e.g., if a processing tool produces composite objects of various data and metadata elements, effectively providing a third dimension. Thus we may end up with all kinds of combinations of compositing, versioning, and provenance relations.

## 4 DISCUSSION OF THE USE CASES

In conclusion, we can see that several ADTs may be used in different ways. Some ambiguity still exists, resulting mainly from the use cases not being detailed enough to decide the important trade-offs and requirements, such as strict ordering and the varying arity of relations. Still, we can conclude that the use cases can at least be partially fulfilled by adequately employing ADTs, and thus it is our goal to provide a framework for such ADTs and a first implementation. To this end, we select Lists, Sets, and Maps as the ADTs for further discussion, given their high relevance to the use cases. Trees and Graph nodes are beyond the scope of this paper; some of their behavior may actually be emulated with Maps, nested Sets, or nested Lists.

An alternative approach to focusing on specific ADTs is to employ graphs for all scenarios as all use case structures can be conceptually represented with graphs that have standardized edge labels, similar to the fundamental data model of Linked Data and the Semantic Web in general (Heath & Bizer, 2011). Such a solution has also been used for SCOPE (Cheung, Hunter, Lashtabeg, & Drennan, 2008), which provides compound objects over scientific data that can be published as named graphs (Carroll, Bizer, Hayes, & Stickler, 2005).

A graph-based approach will however negatively impact the cost of operations as it neglects the optimization potential of specialized ADTs. Obviously, this is an important trade-off, where we trade the flexibility of a graph model for optimized performance on more rigid structures in view of scalability and automated data management.

In addition to this core set of ADTs to implement, we also see four requirements **R1**-**R4** that affect the operations possible on ADTs and the interactions between their instances.

**(R1) Multiple membership:** *Every Digital Object may be an element of two or more ADT instances at the same time. These ADT instances may be of differing types.*

Combining the use cases has shown how an individual Digital Object may become an element of two or more ADT instances at the same time, for example, with two dimensions of provenance, where one is described through a List or Set and the other one through a Map.

**(R2) Nesting:** *Any ADT instance can be nested as an element in any other ADT instance.*

We must ensure that all Digital Objects can act as elements of some ADT instance as well as become the pivot object for another ADT instance at the same time. For example, a List used to link object versions can become part of a Set that describes a composite archive object, as seen when combining use cases. Lacking a full Tree ADT, we need nested sets to fulfill the composition use case.

**(R3) Dedicated PID:** *Every ADT instance must be identified through a dedicated PID.*

We must be able to not only identify each individual element but also have a *head PID* for the ADT instance as a whole. This is relevant to some of the use cases, particularly regarding the master copy in the replication use case, but it is also well suited for identification of experiments or groups in the WDCC archive hierarchy or the latest version of a dataset. Additionally, it offers a neat way to perform the actual ADT operations in a precisely defined context. We therefore are strongly motivated to think of PIDs as the main pivot element of addressing not just Digital Objects but also informally the actual ADT instances and their operations.

If there is no resource associated with such a head PID, it well resembles the definition given in Kahn and Wilensky (2006) for a *meta-object*: “its primary purpose is to provide references to other digital objects”. In this sense, a meta-object PID can, for example, informally identify a List instance.

If the head PID identifies both the ADT instance with its depending elements and a resource, questions arise regarding the relationship between this resource and the elements’ resources. One possible view is that the head resource is the union of all member resources. The replication use case hints at a different usage scenario: the master copy is explicitly not the union of all replicas. We conclude that the detailed semantics are a matter of policy and cannot be provided solely through code, yet the general usefulness of a head PID is clear.

**(R4) Navigability:** *It must be possible to navigate from any ADT instance member PID to any other member PID and also to the dedicated head PID and vice versa.*

One common theme throughout the use cases is the demand for navigability in more than just one direction through an ADT instance and its elements. We can see that Lists used for versioning should allow for navigating from each List element to the ADT instance identifier and also to all other elements. Composite archive objects undoubtedly require that the whole hierarchy be navigable upwards and downwards; thus when using Sets we should be able to navigate both from whole to part as well as from part to whole. We should also always be able to navigate from predecessor to successor objects and vice versa, thus requiring navigability in Maps from all mapped elements to all other elements. Because there may be additional context information available from a provenance head PID, it should be reachable as well.

In the following section, we will provide actual implementations for the selected ADTs.

## 5 IMPLEMENTATION WITH HANDLE SYSTEM RECORDS

Various PID solutions exist and are used by different communities today. Although targeting the Earth Sciences domain, the overview given by Duerr, Downs, Tilmes, Barkstrom, Lenhardt, Glassy, et al. (2011) is largely applicable to other disciplines as well. For this work, we have chosen the Handle System as the practical solution. There are various reasons for this choice, among other things its standardized protocol framework given in three RFCs and its maturity and adoption in practice, e.g., by the International DOI Foundation (http://www.doi.org).

Handles, like other PID solutions, establish a layer of redirection, decoupling the location of a resource (typically a URL) from the additional identifier. Unlike some other PID solutions, the Handle System provides a mechanism to store the additional information associated with a PID close to the identifiers in the form of the Handle record (Handle key-metadata). According to the Handle data model (Sun, Reilly, & Lannom, 2003), each Handle resolves to a number of typed key-value pairs or, more precisely, triples of *(Index, Type, Value)*, where the Index is an unsigned 32 bit integer and Type and Value are arbitrary data. In the following, we will refer to these items with capitalized nouns and to the full set of triples associated with a single Handle as the *Handle record*.

A Handle intrinsically supports two basic resolution operations (Weigel, Lautenschlager, Toussaint, & Kindermann, 2013): retrieving the resource location and retrieving the additional information. It can also fulfill the requirements given in Weigel et al. (2013) from a technical perspective. In total, Handle records provide the foundation required for an implementation of the Persistent Entity concept.

Obviously, we can now use the Handle record to store ADT instance information, or in terms of a PE, use the PE metadata operations to do so. This also means that the ADT instance structure can be preserved even if the identified data are gone. An alternative approach is to use a separate database, which is also a valid solution when using one of the other identifier systems. It must be emphasized that it is possible to implement PEs using such alternatives if seen from the purely functional perspective of enabling all PE operations. However, not all of them are as easily able to fulfill all underlying requirements.

We will present a solution that provides actual implementations of the selected ADTs Set, List, and Map. For the List ADT, we present the two well-known alternatives of Linked Lists and Arrays. For the Map ADT, we establish a hash map structure for efficient lookup operations in constant time. The Set ADT is easily implemented through a hash map as well.

While encoding ADT implementation information in Handle records is technically not a large issue, the more severe challenge is to do so across different authorities that generate Handles. An implementation with the Handle system has two fundamental options for standardization: through the Index or the Type. The Type field has the advantage of holding proper strings as opposed to a single integer value. This provides a larger value space and gives more flexibility to users who wish to deposit additional information in a Handle record. The Index has the advantage of acting as a good primary key for the back-end storage if combined with the Handle name. We see this as an advantage in terms of efficient read and write operations as we can assume mean constant access time through such a combined primary key. The database schemas suggested in the Handle Tech Manual conform to this. Because Type information is not interpreted by the implementation, at this point we only suggest that it should be somewhat standardized, for example, through a controlled vocabulary that helps to document the storage structure of the ADT instances.

Our focus is thus on an index-based solution. We need to take care that type-based arbitrary additional information is tolerated as well to remain compatible with legacy information and other solutions that do not conform to this standardized schema.

## 5.1 Handle Index usage

Sun et al. (2003) define the Handle Index as an unsigned 32 bit integer. We divide the Index into two main segments, segment mask (upper 8 bits) and payload (lower 24 bits) as shown in Figure 2.

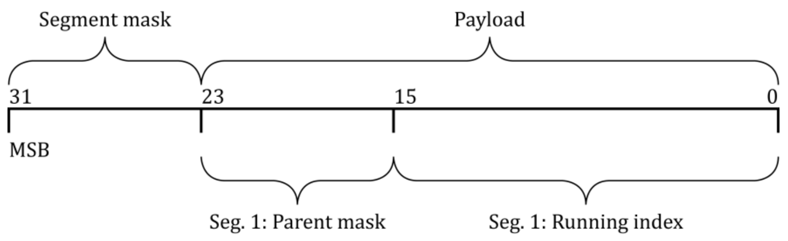

**Figure 2.** 32 bit Index segmentation. Generic segmentation is shown at the top, special payload sub-segments for a segment of type 1 are shown at the bottom. MSB indicates the most significant bit.

In general, the Values at all Indexes with a specific segment mask value will contain the core structural information of one specific implementation. Table 1 gives an overview on the meaning of particular segment mask values.

**Table 1.** Reference for segment mask values

| Mask value | Meaning |
|---|---|
| 0 | Generic information |
| 1 | Parent segment |
| 2 | Array implementation |
| 3 | Hash map implementation |
| 4 | Linked list node implementation |

We will need some generic information entries at fixed Indexes. However, most generic information indexes remain unstandardized, providing the desired compatibility with existing Handle Records and type-based approaches. As a side effect, the fixed Indexes also allow a service to easily detect which implementations are present in a specific Handle Record.

Note that the current implementation of the Handle System appears to use signed but strictly positive integers, making the most significant bit (MSB) unusable. The effective segment mask length is thus reduced to 7 bits as is its referential usage in the parent segment.

### 5.1.1 Parent segment

The parent segment is unique in that its payload bits are further subdivided into a parent mask (8 bits) and a running index (16 bits) - see Figure 2, bottom half. The Value contains the Handle names (head PIDs). Different parent Handles are assigned to particular Indexes depending on what the parent ADT instance type, the parent mask value, is. The parent mask uses the values given in Table 1. Since one object can be an element of multiple ADT instances of the same type, the running index is necessary. To give a complete example: a parent mask of 4 and a running index of 2 are used for the third Linked List of which this Handle is an element. Whenever a Handle is added to an implementation, its parent segment will receive a new entry pointing to the head Handle.

This also means that any add, insert, or remove operations on the individual ADT implementations as well as lookup queries asking for the parent of a particular Digital Object will occur at $O(m)$ additional costs with $m$ being the number of parent ADT instances of same type. This trade-off between efficient operations on single ADT instances and scenarios where an individual element is part of a lot of ADT instances should work out well for the use cases explained previously though it is still a trade-off to be aware of.

### 5.1.2 Array implementation

An Array instance is fully contained within a single Handle record.

Payload values are used in a straightforward way as running indexes. Insert and delete operations will affect up to $n$ entries as entries must be shifted to maintain the strict ordering. Append operations affect only 1 entry. The maximum size of an array is $2^{24}$-1. One generic information entry at a fixed Index (e.g., at Index 2000) stores the current size of the array in its Value.

### 5.1.3 Linked list implementation

A linked list's head Handle contains a Value to the first entry's Handle and another Value to the last entry's Handle at specific fixed Indexes (e.g., at 3001 and 3002) in the generic information section. It also contains a Value storing the current total number of elements in the list (e.g., at Index 3000).

When a Handle is added or inserted into a linked list, its linked list node payload will receive two new entries. To determine their actual Index, the lowest free entry in the parent information sub-segment for linked lists is chosen, $b$, and its Value is set to point to the head Handle. Then the linked list payload entries $2b$ and $2b+1$ are set to have their Values point to the predecessor and successor Handle respectively (also see Table 3). When modifying the first or the last element, the Value is left empty and the generic information in the head Handle record is updated accordingly.

Not counting the *O(m)* costs for parent segment management, this implementation supports insert, append, and remove operations in constant time and efficient iteration in both directions. Iteration efficiency will degrade if elements are part of multiple Linked Lists because the caller will have to access several entries in the parent section to determine the correct set of *2b* and *2b+1*. Indexed access is naturally expensive.

Multiple linked lists can be easily implemented at the expense of the total number of lists an individual object can be member of. An alternative implementation without a dedicated head Handle is also possible; however, we still need a distinct element (e.g., the first element) to fulfill **R1** because otherwise it is impossible to tell what the next or previous element is in the context of a particular list.

As a side note, the current practice of the EUDAT PID services to bind replicas together roughly resembles a linked list implementation although the indexes are not as much standardized there and no effort is made to integrate other ADT implementations without conflicting overlaps.

#### 5.1.4 Hash map implementation

The hash map implementation uses the 24 payload bits as the bucket space of open addressing hash mapping with simple linear probing.

For the Set ADT, the input strings for hashing are proper Handle identifiers, which are typically very short strings that may differ only minimally from each other. One important requirement for the hash function is thus that it does not bear a high collision risk on very small self-similar input data.

For the Map ADT, the input strings can be arbitrary as they are custom user keys, which may but need not be short alphanumeric strings (e.g., from a controlled vocabulary).

There is of course quite some potential for optimization of this hash map implementation in terms of the choice of the hash function and the bucket usage and collision resolution algorithm. For now, we consider this to be future work as lookup and insert time efficiencies are constant already with the presented solution for the average case where the load factor of the whole hash map is negligible due to just a couple of thousand entries.

### 5.2 Practical walk-through

To illustrate the use of the Index for the various ADT implementations, we will now go through a small example. We will use the following pseudo-code fragment, where *a* and *b* are arbitrary Digital Objects, identified by Handle names “100/a” and “100/b”, *map1* and *map2* are hash map implementations, identified by “100/map1” and “100/map2”, *array* is an array implementation, identified by “100/array”, and *linkedlist* is a Linked List implementation, identified by “100/linkedlist”:

```
map1.add(a)
map2.add(a)
map1.contains?(a)
array.append(a)
a.get_parents(hashmap)
linkedlist.append(a)
linkedlist.append(b)
```

Let us assume that the array already contains 17 entries. The hash maps will be empty. The hash function *hash(a)* maps PID names to Integer values between 0 and $2^{23}$-1. The operations then translate individually to:

```
map1.add(a):
  Set Index $3·2^{23}$ + hash(a) on 100/map1 to Value “100/a”
  Get Index $1·2^{23}$ + $3·2^{15}$ + 0 on 100/a
    will return: no value
  Set Index $1·2^{23}$ + $3·2^{15}$ + 0 on 100/a to Value "100/map1"

map2.add(a):
  Set Index $3·2^{23}$ + hash(a) on 100/map2 to Value "100/a"
  Get Index $1·2^{23}$ + $3·2^{15}$ + 0 on 100/a
    will return: occupied with Value "100/map1"
  Get Index $1·2^{23}$ + $3·2^{15}$ + 1 on 100/a
    will return: no value
  Set Index $1·2^{23}$ + $3·2^{15}$ + 1 on 100/a to Value "100/map2"

map1.contains?(a):
  Get Index $3·2^{23}$ + hash(a) on 100/map1
    will return: occupied with Value "100/a"
  Thus return: TRUE

array.append(a):
  Get Index 2000 on 100/array
    will return: occupied with Value "17"
  Set Index $2·2^{23}$ + 17 on 100/array to Value "100/a"
  Set Index 2000 on 100/array to Value "18"
  Get Index $1·2^{23}$ + $2·2^{15}$ + 0 on 100/a
    will return: no value
  Set Index $1·2^{23}$ + $2·2^{15}$ + 0 on 100/a to Value "100/array"
```

Every add or append operation will first write information to the ADT instance and then record membership information in the parent segment of the added object. This causes a second iteration when the element *a* is appended to the second hash map. This is a general case: if an element is added to multiple ADT instances of the same type, looping is required with *O(m)* costs. This effect can also be seen if the parents of the element *a* are queried, even if the particular function knows that, e.g., only hash maps are asked for:

```
a.get_parents(hashmap):
  Get Index $1·2^{23}$ + $3·2^{15}$ + 0 on 100/a
    will return: occupied with Value "100/map1"
  Get Index $1·2^{23}$ + $3·2^{15}$ + 1 on 100/a
    will return: occupied with Value "100/map2"
  Get Index $1·2^{23}$ + $3·2^{15}$ + 2 on 100/a
    will return: no Value
  Thus return: "100/map1", "100/map2"
```

Table 2 lists the relevant Handle Record entries for the affected Handles after the code has been executed.

**Table 2.** Handle Record excerpts (Type omitted). The Index 2000 at Handle 100/array records the current size of the array (18).

| Handle | Index | Value |
|---:|---|---|
| 100/a | $1·2^{23} + 2·2^{15} + 0$ | 100/array |
| 100/a | $1·2^{23} + 3·2^{15} + 0$ | 100/map1 |
| 100/a | $1·2^{23} + 3·2^{15} + 1$ | 100/map2 |
| 100/map1 | $3·2^{23}$ + hash(a) | 100/a |
| 100/map2 | $3·2^{23}$ + hash(a) | 100/a |
| 100/array | 2000 | 18 |
| 100/array | $2·2^{23} + 17$ | 100/a |

We will now have a separate look at the Linked List implementation. In the following code fragment, an initially empty Linked List instance "100/linkedlist" is appended consecutively with two elements "100/a" and "100/b":

```
linkedlist.append(a):
  Get Index 3000 on 100/linkedlist
    will return: occupied with Value "0"
  Set Index 3000 on 100/linkedlist to Value "1"
  Set Index 3001 on 100/linkedlist to Value "100/a"
  Set Index 3002 on 100/linkedlist to Value "100/a"
  Get Index $1\cdot2^{23}$ + $4\cdot2^{15}$ + 0 on 100/a
    will return: no Value
  Set Index $1\cdot2^{23}$ + $4\cdot2^{15}$ + 0 on 100/a to Value "100/linkedlist"

linkedlist.append(b):
  Get Index 3000 on 100/linkedlist
    will return: occupied with Value "1"
  Set Index 3000 on 100/linkedlist to Value "2"
  Get Index 3002 on 100/linkedlist
    will return: occupied with Value "100/a"
  Set Index 3002 on 100/linkedlist to Value "100/b"
  Get Index $1\cdot2^{23}$ + $4\cdot2^{15}$ + 0 on 100/b
    will return: no Value
  Set Index $1\cdot2^{23}$ + $4\cdot2^{15}$ + 0 on 100/b to Value "100/linkedlist"
  Get Index $1\cdot2^{23}$ + $4\cdot2^{15}$ + 0 on 100/a
    will return: occupied with Value "100/linkedlist"
  Set Index $4\cdot2^{23}$ + 1 on 100/a to Value "100/b"
  Set Index $4\cdot2^{23}$ + 0 on 100/b to Value "100/a"
```

Table 3 lists the relevant Handle Record entries for the affected Handles after the linked list code has been executed. The full record would be the combination of Table 2 and Table 3. If the element *a* were a member of several linked lists and had predecessors and successors in them as well, then it would also have Values for Indexes such as $4\cdot2^{23} + 2$, $4\cdot2^{23} + 3$ and higher.

**Table 3.** Commented Handle Record excerpts (Type omitted).

| Handle | Index | Value | Comments |
|---|---|---|---|
| 100/a | $1\cdot2^{23} + 4\cdot2^{15} + 0$ | 100/linkedlist | The parent element |
| 100/a | $4\cdot2^{23} + 1$ | 100/b | *2b+1*, i.e. the next element |
| 100/b | $1\cdot2^{23} + 4\cdot2^{15} + 0$ | 100/linkedlist | The parent element |
| 100/b | $4\cdot2^{23} + 0$ | 100/a | *2b*, i.e. the previous element |
| 100/linkedlist | 3000 | 2 | Current number of list elements |
| 100/linkedlist | 3001 | 100/a | The first list element |
| 100/linkedlist | 3002 | 100/b | The last list element |

## 5.3 Discussion and future work

We will briefly discuss how the four requirements are fulfilled by the presented solution.

The parent segment can hold many references, independent of the parent instance type. A linked list node is also able to distinguish between multiple memberships in several linked lists. **R1** is thus fulfilled.

Because information about parents and members is split between different segments and because segments in general do not interfere with each other due to the bit mask, **R2** is fulfilled. We have designed every implementation with a dedicated element in mind to conform to **R3**. The parent segment is the key mechanism to fulfill **R4**: full navigability is typically achieved by going through the head PID. Linked lists also support efficient navigation between direct list neighbor elements. The solution is also extensible due to the facilities of segment mask and arbitrary payload.

One important restriction is that while one Handle can act as the head for multiple different implementations, it cannot do so for multiple instances of the same implementation, a problem that can, however, be circumvented through nesting.

One potential issue arises if ADT instances are created with massive numbers of entries, which may cause scalability issues as it is not clear yet what the future development strategy and infrastructural support of the Handle System will be. We can only attempt at this point to make scalability and support for large Handle records a main objective. The data model of the Handle System is very generic so that if basic resolution of PIDs to their data is significantly affected, the more heavyweight Handle records might be externalized. We address this by emphasizing the interface principle of ADTs, which can ultimately also transparently support solutions that store ADT instance information in separate metadata databases. From a user's viewpoint, it should not matter where the actual instance information is stored and whether resource resolution and metadata resolution operations are implemented through different pathways.

The individual entities that are bound together through ADTs may carry additional context information in their Handle records. Although this information is interpreted by higher layers and is opaque to ADT operations, there is nonetheless a gain in discoverability and navigability for the additional information. To enable machine-interpretability and interaction with these elements as well as with the ADTs as a whole, this information should be typed, and the types should be registered.

Looking at possible applications for the presented ADTs, we find the iRODS system to be a promising candidate. Some of the core entities managed within iRODS can be adequately aligned to the presented ADTs. The most straightforward examples from the iRODS world include files and workflows, identified by single PIDs, and collections, replicas, versions, and backups, identified by list ADTs and with their content (usually files) individually identified. But there are also more complex examples that raise interesting questions.

In iRODS, there are various usage scenarios for soft links that point to objects in external systems. Such an external object may already bear its own PID. If it is registered within the local iRODS instance and perhaps put in a local collection, the question is whether it will receive a new PID and whether that PID depicts the target object or rather the membership of the object in the local collection. More generally, we must ask: Does the PID denote the entity or the registration of the entity in a new namespace? In terms of ADTs, this could be solved, for example, via membership in a List: The PID of the external file could be added to a local List instance, and due to **R1**, this should not affect any additional structural membership at the external site. This, however, requires interoperability between the PID systems and namespaces and also reveals possible security concerns that must be addressed first.

There are also various scenarios that call for the use of extra information, which types the PID or the object behind it, for example, to distinguish between proper external files and only cached copies thereof, which has implications for the operating rules of iRODS instances. A user of the ADT instances will need to have type information associated with it to understand how to manipulate the contents. The current ADTs are ignorant of the potential typing of their members; most of all, an ADT instance such as a List does not assume that all its members are of same type. But if we introduce types, how does this impact the ADT operations and an agent's interpretation of List membership? With the possibility to nest ADTs, information relevant for interpretation could be included in the form of sub-lists. But then, how should this nested information be interpreted, based on typing, and with possible other data-related sub-lists present? And how do we minimize the number of recursive calls to gain the required information for correct interpretation? Overall, the combination of typing and ADT instances can provide important benefits and solutions, but it also presents challenges not yet addressed.

Interesting scenarios also emerge from the management and use of time-series data. Access operations on dynamic time-series data may be very individual as each request can slice off different parts of the series. If such individual responses receive PIDs, then we must ask how these PIDs should be set up so they properly identify the operations that took place. With ADT instances such as Lists available, we must ask: Can such a structured response PID be interpreted as an operation on a List of PIDs with associated input parameters? Is the identification of such operations that are performed upon the collection a better way to characterize the use of structured objects such as time-series data collections? If every slicing operation must be persistently identified, scalability issues may emerge, and a more dynamic approach to collection operations could be highly useful.

Answering such questions will help to further define the practical use and benefits of actionable PID collections, but it requires further modeling and thought experiments going beyond the scope of this article. In particular the issues around typing are a matter of RDA working group activities and can eventually lead to consolidation.

## 6 RELATED WORK

There are many other PID systems or infrastructures besides the Handle System that are potential candidates for implementation. DataCite DOIs form a practically adopted solution for persistently identifying data objects used in academic practice with a focus on long-term archival and citability. Technically, the DOI System is an application of the Handle System, and every DOI is in fact a Handle. In addition, however, the DOI System and DataCite provide layers on top of the Handle System such as standardized metadata. The DataCite metadata kernel 2.2 offers possibilities to emulate some of the fundamental ADT implementations, such as sets or linked lists through part-of/is-part-of and predecessor/successor relations, so it may be used in an alternative implementation. However, no unified ADT operations exist within DataCite services, and the preservation of PEs without preserving the actual data is out of the DataCite scope.

The ARK system (Kunze, 2003) provides actionable identifiers through special suffixes, which can be appended when resolving an ARK identifier. Most significantly, a metadata record can be retrieved separately from a resource. In combination, this may well serve the purpose of a PE. Given a consistent metadata schema, ARKs can support ADT operations though it is unclear what the method efficiency will be.

There is a potential overlap with Linked Data (see, e.g., Heath & Bizer, 2011, Bizer, 2013). Its graph-based data model may also provide a valid implementation, however, with less optimization potential regarding efficiency. Nonetheless, we can see how Linked Data can form a higher-level layer, exposing the information implicit through the ADT relations in an RDF encoding. Obviously, for the relations implicit to the presented ADTs, suitable equivalent elements in the Linked Data world would have to be found. To gain a richer level of informational detail and finally enable reasoning capabilities, sources other than the ADTs must be integrated. We also see a distinct and intended division of work and scope here. The lower layer focuses on breadth and coverage yet offers only a shallow level of detail while the Linked Data layer provides more detailed information, which is also more difficult to acquire and maintain. In this respect, the lower layer can act as a rudimentary fallback position.

Another example for a higher-level layer is the idea of Research Objects (Bechhofer, Ainsworth, Bhagat, Buchan, Couch, Cruickshank, et al., 2010), which are identifiable context-rich compound objects for scientific data that also include provenance information. A similar example is the SCOPE approach (Cheung et al., 2008), which itself is based on the Open Archive Initiative Object Reuse and Exchange protocol (OAI-ORE). OAI-ORE offers an aggregation concept through "Resource Maps" (cf. http://www.openarchives.org/ore/1.0/datamodel). In OAI-ORE Resource Maps, methods exist to construct linked lists and unordered sets, and there are also proxy mechanisms that ensure conflict-free membership in multiple aggregations.

The eSciDoc system (Razum, Schwichtenberg, Wagner, & Hoppe, 2009) covers versioning, provenance, and composition in combination with PIDs. CLARIN Virtual Collections (http://www.clarin.eu/sites/default/files/virtual_collections-CLARIN-ShortGuide.pdf) serve a purpose overlapping with the composite objects use case. Most notably, they use the Handle System for PIDs though the structural information is stored in a separate metadata database. Although all of these exemplary solutions do not offer a comparatively full set of ADTs, they could be significantly enhanced and improved by adopting them.

## 7 CONCLUSIONS AND OUTLOOK

We have presented a number of use cases that illustrate the practical value and use of PIDs for research data. From there, we have drawn the motivation to establish abstract data types as the fundamental interface enabling automation with PIDs as the main method for addressing domain objects. Our practical solution based on the Handle System offers a set of exemplary ADT implementations and thereby provides actionable *sets-of-handles*. The implemented ADTs are Lists, Sets, and Maps. One important design principle is following a layered architecture and intentionally limiting functionality. As a result, the ADT instances do not need to deal with strong semantics and can also remain ignorant of the content of the actual entities they organize, which is particularly important for automated data management tasks. The solution is designed to be extensible within the scope of its layer, for example, by including implementations of other ADTs in the future. In accordance with the goals of the Persistent Entity concept, the structural information of ADT instances can be preserved even if the identified data object is gone.

Our solution uses Handle records and is thus very much tailored to one particular PID infrastructure. Depending on ADT usage in practice, scalability issues may arise if future strategic development of the Handle System infrastructure focuses on its core competency of identifier resolution to resource locations and aims to keep the PID records thin rather than providing extensive additional information. Nonetheless, the conceptual Handle data model remains very generic, and a practical compromise might allow for information to be externalized in more heavyweight metadata records while keeping the ADT API.

There are also many other PID systems, and it is unclear whether a single predominant solution will emerge in the future. Although the presented implementation of ADTs is specific to the Handle System, the underlying concept of ADTs whose operations use PIDs as their main mode of addressing is transferable to other systems. ADTs should become available on top of other PID systems as well, and the fundamental concept of having ADT operations working with PIDs should be the pivot point for interoperability across these different systems. As with the relationship of ADTs and particular implementations, the exemplary implementation is bound to its platform, but the interface is formed by ADTs that abstract from that platform and can be implemented elsewhere as well. In the end, obviously, it should then be possible, for example, to add PIDs from different systems to a single list, provided that permission and security systems are compatible. Moreover, there is a high potential to leverage and enhance existing solutions that exist in the Linked Data world or in the area of e-science infrastructures, such as eSciDoc and OAI-ORE enabled solutions. A possible application of the presented ADTs within iRODS offers both a promising usage scenario and a number of interesting research questions.

## 8 ACKNOWLEDGEMENTS

Acknowledgements go to the anonymous peer reviewers whose feedback helped to improve the contents of this article.

## 9 REFERENCES

Bechhofer, S., Ainsworth, J., Bhagat, J., Buchan, I., Couch, P., Cruickshank, D., Roure, D. D., Delderfield, M., Dunlop, I., Gamble, M., Goble, C., Michaelides, D., Missier, P., Owen, S., Newman, D., & Sufi, S. (2010) Why Linked Data is Not Enough for Scientists. *e-Science 2010, Sixth International Conference on e-Science*, Brisbane, Australia. Retrieved Nov 11, 2013 from the World Wide Web: http://dx.doi.org/10.1109/escience.2010.21

Bizer, C. (2013) Interlinking Scientific Data on a Global Scale. *Data Science Journal 12*. Retrieved Nov 11, 2013 from the World Wide Web: http://dx.doi.org/10.2481/dsj.GRDI-002

Carroll, J. J., Bizer, C., Hayes, P., & Stickler, P. (2005) Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on the World Wide Web*. Retrieved Nov 11, 2013 from the World Wide Web: http://dx.doi.org/10.1145/1060745.1060835

Cheung, K., Hunter, J., Lashtabeg, A., & Drennan, J. (2008) SCOPE: A Scientific Compound Object Publishing and Editing System. *International Journal of Digital Curation 3*(2), pp. 4-18. Retrieved Nov 11, 2013 from the World Wide Web: http://dx.doi.org/10.2218/ijdc.v3i2.55

Duerr, R. E., Downs, R. R., Tilmes, C., Barkstrom, B., Lenhardt, W. C., Glassy, J., Bermudez, L. E., & Slaughter, P. (2011) On the utility of identification schemes for digital earth science data: an assessment and recommendations. *Earth Science Informatics 4*(3), pp. 139–160. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.1007/s12145-011-0083-6

Groth, P., Gil, Y., Cheney, J., & Miles, S. (2012) Requirements for Provenance on the Web. *International Journal of Digital Curation 7*(1), pp. 39-56. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.2218/ijdc.v7i1.213

Heath, T. & Bizer, C. (2011) Linked Data: Evolving the Web into a Global Data Space. *Synthesis Lectures on the Semantic Web: Theory and Technology 1*(1). Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.2200/s00334ed1v01y201102wbe001

Hey, T., Tansley, S., & Tolle, K. (Eds.) (2009) *The fourth paradigm: data-intensive scientific discovery*. Redmond, WA: Microsoft Research.

ISO/IEC 9834-8:2008, Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components. International Standards Organization.

Kahn, R. & Wilensky, R. (2006) A framework for distributed digital object services. *International Journal on Digital Libraries 6*(2), pp. 115–123. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.1007/s00799-005-0128-x

Kunze, J. A. (2003) Towards Electronic Persistence Using ARK Identifiers, ARK motivation and overview. In *Proceedings of the 3rd ECDL Workshop on Web Archives.* Trondheim, Norway.

Liskov, B. & Zilles, S. (1974) Programming with abstract data types. *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages 9*(4), 50-59. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.1145/800233.807045

Moreau, L. (2010) *The Foundations for Provenance on the Web, Foundations and Trends® in Web Science, 2*(2-3), pp. 99–241, Delft: now Publishers.

Moreau, L. & Missier, P. (Eds.) (2013) *PROV-DM: The PROV Data Model.* W3C Recommendation. Retrieved Nov 11, 2013 from the World Wide Web: http://www.w3.org/TR/2013/REC-prov-dm-20130430/

Moreau, L., Ludäscher, B., Altintas, I., Barga, R. S., Bowers, S., Callahan, S., Chin, G., Clifford, B., Cohen, S., Cohen-Boulakia, S., Davidson, S., Deelman, E., Digiampietri, L., Foster, I., Freire, J., Frew, J., Futrelle, J., Gibson, T., Gil, Y., Goble, C., Golbeck, J., Groth, P., Holland, D. A., Jiang, S., Kim, J., Koop, D., Krenek, A., McPhillips, T., Mehta, G., Miles, S., Metzger, D., Munroe, S., Myers, J., Plale, B., Podhorszki, N., Ratnakar, V., Santos, E., Scheidegger, C., Schuchardt, K., Seltzer, M., Simmhan, Y. L., Silva, C., Slaughter, P., Stephan, E., Stevens, R., Turi, D., Vo, H., Wilde, M., Zhao, J., & Zhao, Y. (2008) Special Issue: The First Provenance Challenge. *Concurrency and Computation: Practice and Experience 20*(5), pp. 409–418. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.1002/cpe.1233

Paskin, N. (2003) Components of DRM Systems, Identification and Metadata. In Becker, E., Buhse, W., Günnewig, D., & Rump, N. (Eds.), *Digital Rights Management, Lecture Notes in Computer Science 2770*. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.1007/10941270_4

Paskin, N. (2009) Digital Object Identifier (DOI) System. In Bates, M. J. & Maack, M. N. (Eds.), *Encyclopedia of Library and Information Sciences, Third Edition*. Taylor & Francis.

Rajasekar, A., Moore, R., Hou, C.-Y., Lee, C. A., Marciano, R., de Torcy, A., Wan, M., Schroeder, W., Chen, S.-Y., Gilbert, L., Tooby, P., & Zhu, B. (2010) iRODS Primer: Integrated Rule-Oriented Data System. *Synthesis Lectures on Information Concepts, Retrieval, and Services 2*(1), pp. 1-143. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.2200/s00233ed1v01y200912icr012

Razum, M., Schwichtenberg, F., Wagner, S., & Hoppe, M. (2009) eSciDoc Infrastructure: A Fedora-Based e-Research Framework. In Agosti, M., Borbinha, J., Kapidakis, S., Papatheodorou, C., & Tsakonas, G. (Eds.), *Research and Advanced Technology for Digital Libraries, Lecture Notes in Computer Science vol. 5714*, pp. 227-238. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.1007/978-3-642-04346-8_23

Sollins, K. & Masinter, L. (1994) *RFC 1737: Functional requirements for uniform resource names.* IETF.

Sun, S., Reilly, S., & Lannom, L. (2003) *RFC 3651: Handle system namespace and service definition*. IETF.

Weigel, T., Lautenschlager, M., Toussaint, F., & Kindermann, S. (2013) A Framework for Extended Persistent Identification of Scientific Assets. *Data Science Journal 12*, pp. 10-22. Retrieved Nov 11, 2013, from the World Wide Web: http://dx.doi.org/10.2481/dsj.12-036

(Article history: Received 11 November 2013, Accepted 22 December 2013, Available online 10 January 2014)