Digital Preservation Based on Contextualized Dependencies

Nikolaos Lagos
Xerox Research Centre Europe
6 chemin de Maupertuis
Meylan, France
0033(0)476615192
nikolaos.lagos@xrce.xerox.com

Jean-Yves Vion-Dury
Xerox Research Centre Europe
6 chemin de Maupertuis
Meylan, France
0033(0)476615152
jean-yves.vion-dury@xrce.xerox.com

ABSTRACT
Most of existing efforts in digital preservation have focused on extending the life of documents beyond their period of creation, without taking into account intentions and assumptions made. However, in a continuously evolving setting, knowledge about the context of documents is nearly mandatory for their continuous understanding, use, care, and sustainable governance. In this work we propose a method that considers the preservation of a number of interdependent digital entities, including documents, in conformance with context related information. A change that influences one of these objects can be propagated to the rest of the objects via analysis of their represented dependencies. We propose to represent dependencies not only as simple links but as complex, semantically rich, constructs that encompass context-related information. We illustrate how this method can aid in fine-grained contextually-aware change propagation and impact analysis with a case study.

Keywords
Digital preservation; Dependency; Data Governance; Context; Linked Resource Model.

1. INTRODUCTION
Corporations, government agencies and other organizations dedicate a big amount of effort for defining document and data governance strategies to help them contend with fast-growing and diverse pools of documents, saved in a variety of formats. The series of managed activities that ensure continued access or renew the usability of documents, and in general digital information, for as long as necessary, is coined digital preservation [1], [2].

Most of existing efforts in digital preservation have focused on extending the life of documents beyond their period of creation without taking into account initial intentions and assumptions made. However, in a continuously evolving setting, knowledge about the context of documents is nearly mandatory for their continuous understanding, use, care, and sustainable governance [3].

Researchers and experts from diverse domains have highlighted the significance of context in digital preservation. [4] when studying the preservation of software documents (including software programs) argues that there would be significant long-term benefits if specification documents supported wider questions about the reasons why certain approaches were adopted. He proposes recording the context of intended use by creating what he calls intent specifications documents. [5] notes that poor maintenance is often a contributory factor in software induced accidents.

Interestingly, the National Library of Australia’s Digital Collections has also started using recently what they call “preservation intent statements” i.e. records of the preservation context, including initial intent, for specific classes of digital content, such as digital documents. They argue that this also relates to the concept of “significant properties” i.e. properties that define the (not necessarily strictly quantifiable but observable) qualities that the preserved object has to adhere to, some of these being related to its context [6].

Being able to identify how changes can affect the context of preserved documents is especially crucial in environments that are subject to continual change. Recently, corresponding preservation models consider the separation of preservation from active life as not feasible or desirable [7]. In such cases, to maintain the reusability of complex digital objects and their associated environment, it is necessary to consider risks that can occur due to changes in the environment, to allow determining and performing appropriate mitigating actions.

In this work we propose a method that considers the preservation of a number of interdependent digital entities, including documents, in conformance with context related information. Such entities are assumed to exist within a continually changing environment, which may result in them becoming unusable over time. A major aspect of the methodology is viewing the collection of digital entities as a network of objects where the links between them represent their dependencies. A change that influences one of these objects can then be propagated to the rest of the objects via an analysis of the represented dependencies. We propose to represent dependencies not only as simple links but as complex, semantically rich, constructs that encompass context-related information. In that manner, the corresponding change propagation algorithms take into account the context, encoded in dependency-specific properties.
In the next section we introduce the main concepts we use to represent digital entities, their dependencies, and the corresponding context. To illustrate a concrete representation of these concepts, we developed an ontology (encoded in OWL2) that semantically defines them in a formal manner. In Section 3 we present how dynamic information is recorded and acted upon via a change propagation algorithm that is conditioned on the represented context parameters. A case study is described in Section 4 that demonstrates the advantages of the approach. A review of related work is included in Section 5 and conclusions and directions for future work in Section 6.

2. REPRESENTATION MODEL

As the first step in our methodology we define a model that allows us to represent the preservation ecosystem (i.e. all objects to be preserved and their interconnections) in terms of the corresponding context. To achieve that we define the Linked Resource Model (LRM), as described in the remaining of the section.

The LRM is an upper level ontology designed to provide a principled way to modelling evolving ecosystems, focusing on aspects related to the changes taking place\(^1\). This means that, in addition to existing preservation models that aim to ensure that records remain accessible, the LRM also aims to model how changes to the context, and their impact, can be captured. It is important to note here that we assume that a change policy governs at all times the dynamic aspects related to changes (e.g. conditions required for a change to happen and/or impact of changes). As a consequence, the properties of the LRM are dependent on the change policy being applied; therefore, most of the defined concepts are related to what the policy expects. At its core the LRM defines the ecosystem by means of participating entities and dependencies among them. A set of other properties and specialised entity types are also provided but they are all conditioned on what is allowed/required by the change policy. The main concepts of the LRM are illustrated in Figure 1 and discussed further below.

2.1 Resource

The concept of Resource in LRM represents any entity in the universe of discourse of the LRM Model. A resource can be Abstract (c.f. AbstractResource in Figure 1), representing the abstract part of a resource, for instance the idea or concept of a book, or Concrete (c.f. ConcreteResource in Figure 1), representing the part of an entity that has a physical extension and can therefore be accessed at a specific location (a corresponding attribute called location is used to specify spatial information; for instance for a Digital-resource, which represents entities with a digital extension, this information can be the URL required to retrieve and download the corresponding bit stream). The above two concepts can be used together to describe an entity; for example, Don Quixote, as referred by documents talking about the ideas or intentions of the novel, and a corresponding realisation of the novel as a specific book that can be read in the library. To achieve that, the abstract and concrete resources can be related through a specific realizedAs predicate.

2.2 Dependency

The core concept of the LRM is that of a dependency (Figure 2).

An LRM Dependency describes the context under which change in one or more LRM resources has an impact on other LRM resources.

For instance, consider a document containing a set of diagrams that has been created using MS Visio 2000, and that a corresponding policy defines that MS Visio drawings should be periodically backed up as JPEG objects by the work group who created the set of diagrams in the first place\(^2\).

---

\(^1\) The ontology can be downloaded at http://download.xrce.xerox.com/doceng2016/doceng2016_Appendix.zip

\(^2\) This example is adapted from a use case described in [8], pp. 52-53.
According to the policy, the work group who created the set of JPEG objects should be able to access but not edit the corresponding objects. We should be able to use the classes and properties related to the Dependency class to describe each such conversion in terms of its temporal information and the entities it involves along with their roles in the relationship (i.e. person making the conversion and object being converted).

The description of a dependency should therefore minimally include the intent or purpose related to the corresponding usage of the involved entities. From a functional perspective, we expect that dedicated policies/rules should further refine the context (i.e. conditions, impacts) under which change is to be interpreted for a given type of dependency.

Explicit classes are also defined for representing conjunctive and disjunctive dependencies (denoted accordingly as ConjointiveDependency, DisjunctiveDependency). In the case that a ConjointiveDependency is used then if any of the resources related to that dependency, via the lrms:from property, is changed then the dependency is "activated", or in other words the corresponding impact can be computed. Conversely, in the case of DisjunctiveDependency all resources at the source of the lrms:from properties should be changed for a dependency to be activated. Of course, this has repercussions on the inference related to the calculation of the impact and corresponding change propagation within the network of dependencies, as explained in section 3.1.

2.3 Contextualising Dependencies

2.3.1 Intent

The LRM Dependency is strictly defined according to the intent underlying a specific change. In the example described in Section 2.2 the intent may be described as "The work group who created the set of diagrams wants to be able to access (but not edit) the diagrams created using MS Visio 2000. Therefore, the work group has decided to convert these diagrams to JPEG format".

To enable recording the intent of a dependency, LRM relates any Dependency entity with an entity that describes the intent via a property that we name intention, as illustrated in Figure 2. The intent can be described in a formal and/or informal manner i.e. unstructured text. In LRM, the Description class represents informal or formal (i.e. expressed in a formal language) written accounts of information related to an lrms:Resource. In addition to what is shown in Figure 2, we specialise the Description class into a subclass that we call Intention to define the space of all possible intents. The expressed intent implies the following:

1. There is a dependency between the MS Visio and JPEG objects. More specifically, the JPEG objects are depending on the MS Visio ones. This means that if an MS Visio object 'MS1' is converted to a JPEG object, 'JPEG1', and 'MS1' is edited, then 'JPEG1' should either be updated accordingly or another JPEG object 'JPEG2' should be generated and 'JPEG1' optionally deleted (the use case is not explicit enough here to decide which of the two actions should be performed). This dependency would be particularly useful in a scenario where MS Visio keeps on being used for some time in parallel to the JPEG entities, which are in turn used for back up purposes.

2. The dependency between 'MS1' and 'JPEG1' is unidirectional. Actually, JPEG objects should not be directly modified. If they are, no change to the corresponding MS Visio objects should apply.

3. The dependency applies to the specific work group, which means that if a person from another work group modifies one of the MS Visio objects, no specific conversion action has to be taken (the action should be defined by the corresponding change policy).

Some of the above implications, for instance (2), can be expressed by the directionality of the dependency. We define two properties, from and to, that indicate the directionality of the dependency, as shown in Figure 2.

Expressing more complex behaviour like the one described in implication (3) requires another type of representation. We define a notion called specification that defines the expected constraints and properties that the dependency should adhere to (in a sense its expected behaviour) in a formal and/or informal manner. To enable recording the specification of a dependency, we relate in the LRM the Dependency entity with an entity that describes the specification via a property that we name specification, as illustrated in Figure 2. As in the case of intention, we can specialise the Description class into a subclass that we call Specification and which represents the space of all possible specifications.

2.3.2 Preconditions and Impacts

The LRM model provides also concepts that allow the context to be taken into account in an operational manner by recording when a change is triggered and what is the impact of this change on other entities. Let us take once more the above example: we need to be able to express the fact that transformation to JPEG objects is possible only if the corresponding MS Visio objects exist and if the human that triggers the conversion has the required permissions to do that (i.e. belongs to the specific workgroup). The impact of the conversion could be to generate a new JPEG object or update an existing one. The action to be taken (i.e. generate or update) in that case, would be decided based on the change policy governing the specific operation. Assuming that only the most recent JPEG object must be archived, then the old one must be deleted and replaced by the new one (conversely deciding to keep the old JPEG object may imply having to archive the old version of the corresponding old MS Visio object as well). The precondition(s) and impact(s) of a change operation are connected to the Dependency concept in LRM via precondition and impact properties as illustrated in Figure 2. These connect a Dependency to a Plan, which represents a set of actions or steps to be executed by someone/something (either human or software). The Plan can be used, thus, as a means of giving operational semantics to dependencies. Plans can describe how preconditions and impacts are checked and implemented (this could be for example defined via a formal rule-based language, such as SWRL [23]). An example of a Specification is given in Appendix A.

2.3.3 Specialising Dependencies based on Context

One of the strengths of having the dependency notion represented as a class is that we can use the inference mechanisms of OWL to specialize and extend it, according to the intents, specifications, impacts, and preconditions that define it. Take the example of expressing a dependency that defines that the validation of an XML file is dependent on the XML schema and the impact of the validation activity is having a validation report. We can first define the corresponding XML validation dependency as follows:
To specialise this dependency class according to the specific intents, preconditions, and impacts, we can follow a number of approaches, for instance by specializing the Description and Plan classes and then connect the dependency to these specialised classes as shown below (in Turtle [22]).

```turtle
ex:XmlValidation rdfs:subClassOf lrm:ConjunctiveDependency.

ex:XmlValidationDescription rdf:type owl:Class ;
   rdfs:subClassOf lrm:Description .
ex:XmlValidationPreCondition rdf:type owl:Class ;
   rdfs:subClassOf lrm:Plan .
ex:XmlValidationImpact rdf:type owl:Class ;
   rdfs:subClassOf lrm:Plan .
ex:XmlValidation rdf:type owl:Class ;
   rdfs:subClassOf lrm:ConjunctiveDependency;
   owl:equivalentClass [ rdf:type owl:Class ;
   owl:intersectionOf [ [ rdf:type owl:Restriction ;
   owl:onProperty lrm:intention ;
   owl:someValuesFrom ex:XmlValidationDescription ]
   [ rdf:type owl:Restriction ;
   owl:onProperty lrm:precondition ;
   owl:someValuesFrom ex:XmlValidationPreCondition ]
   [ rdf:type owl:Restriction ;
   owl:onProperty lrm:impact ;
```

3. OPERATIONAL ASPECTS

3.1 Interpreting Dependencies

The concepts of ConjunctiveDependency and DisjunctiveDependency were introduced in Section 2.2 (see also [17] for an initial introduction to similar concepts). Below we provide an explanation of the interpretation of these two classes in terms of change propagation in the dependency graph.

- Conjunctive dependencies (ConjunctiveDependency): all source resources are required together and simultaneously to evaluate the impact of a change on the target resource(s). It means that if one or several of the sources has changed (and the preconditions are satisfied), then the target resources must change according to the impact;
- Disjunctive dependencies (DisjunctiveDependency): the target resource(s) do not need to change if there is at least one source resource that did not change.

These two classes of dependencies and their combination allow for expressing the strong conjunctive hypothesis: for instance in Figure 3 an XML document, to be transformed, depends on an XML parser, on a tree-to-tree transformer and on an XML linearizer; but also the fact that a variety of choices for the source resources are possible: for instance the XML parser can be chosen among several standard ones, having slight performance differences, but providing identical functionality, in terms of what is required for the dependency (Figure 4).

![Figure 3: a conjunctive dependency](image)

In Figure 4 the composition of dependencies is done through a particular resource of the class D-Connector. D-Connector has no other particular semantics than being a “virtual” intermediate resource that is used to propagate the change information in a homogenous way, and therefore, simplifying the algorithms presented in the following sections.

![Figure 4: composition of different dependency types](image)

3 This is a problem: after an unsuccessful evaluation of the operation the state of the dependency network would be altered. This hinders, for instance, reproducibility.
only if the corresponding conditions evaluate to true. Context here corresponds to a specific instantiation of the dependency graph that can be implemented, for example, as an RDF graph4 or a triple store $S$.

We propose using a nested transaction mechanism [9] adapted to handle triple stores through a stack-like structure, and based on three standard verbs:

- **TransactionStart($S$)**: Pushes one new entry on the store’s stack; each triple addition or deletion will be recorded in this new entry.
- **TransactionCommit($S$)**: Pops the top level entry out of the stack, and add or withdraw all corresponding triples inside the new top level entry (or updates the ground store if the stack is empty).
- **TransactionAbort($S$)**: Pops the top level entry out of the stack, and does nothing else (that is, forgets everything about the corresponding triples inside the old top level entry).

Reading the store means that the stack is coherently exploited (scanning the stack from top level down to ground level - the basic store), and is abstracted through the mathematical relation $t \in S$, where $t$ stands for an RDF triple.

### 3.3 Context-aware Change Propagation over the Global Dependency Graph

A forward chaining interpretation algorithm can be used to propagate the impact of changes. This is useful when the system is notified of a change to one of the represented resources. In this mode, the change is propagated through the whole representation system by analyzing the dependency graph in a forward fashion.

As preliminaries, we define two functions in Procedure 1 to detect and signal the change of a resource $r$ inside an RDF graph or triple store $S$, through the use of a particular object called lrm:changeMarker. An RDF triple with subject $s$, predicate $p$ and object $o$ is denoted by RDF($s$, $p$, $o$).

The first function, denoted Check-Change($S$, $r$), is a boolean function which checks whether a resource $r$ in a triple store $S$ has been changed (true) or not (false).

The second function, denoted Set-Change($S$, $r$), registers that a change has occurred to the resource $r$ in the store $S$ and inserts the corresponding triple.

```java
boolean Check-Change($S$, $r$) is [return RDF(lrm:changeMarker, lrm:subject, $r$) \in S]
void Set-Change($S$, $r$) is [set RDF(lrm:changeMarker, lrm:subject, $r$) \in S]
```

**Procedure 1:** detecting and signaling the change of a resource $r$ inside a store $S$

As described in section 2, dependencies in LRM are defined according to the intended usage of linked resources, which we capture through the property lrm:intention, that points to the class lrm:Intention. Each dependency instance is therefore associated with a particular intention that is analyzed during the interpretation of the dependency graph. More precisely, a particular dependency will be selected if and only if its intention is compatible with the target intention that characterizes the graph analysis, and which is one of the input parameters. The procedure below (Procedure 2) handles this information through calling the underlying RDF inference mechanism to check for class membership. We note $\exists t \text{ to express that the triple } t \text{ is inferred from the store } S$.

```java
// $S$ is a store, $i$ an Intention descriptor, $i$ is an instance of type Intention, $d$ a dependency instance
boolean Check-Intention($S$, $d$, $i$) is
    return RDF($d$, lrm:intention, $i$) \in $S$ and $S \ni RDF(i, rdf:type, i)$
```

**Procedure 2:** checking that a dependency is compatible with a particular intention

A recursive method, which combines the two Procedures 1 and 2, as illustrated in Algorithm 1 below, explores the whole dependency graph starting from the changed resource. Cycles are eliminated by memorizing the resources already explored inside a set $M$. If a resource is already in the set $M$, this means it will not be considered again, thereby avoiding never-ending cycles to occur. The precondition predicate introduced in Section 2.3.2 is used in Algorithm 3 (e.g. precondition($S$, $d$) and links a dependency $d$ to a logical property that must be evaluated as true for $d$ to be activated. The impact condition is typically expected to change the context (e.g. the RDF store $S$), including the target resource(s). In case $d$ has no to predicate, the impact condition always evaluates to true (or its absence is evaluated as true).

```java
void FWD-Change($S$, $i$, $r$, $M$) is
    // $S$ is a store, $i$ an Intention descriptor, $r$ a resource
    // $M$ is a set to memorize the exploration path
    if ($r \notin M$) {
        set $r \in M$; // $d$ is a dependency
        forall $d$ such that RDF($d$, lrm:from, $r$) \in $S$ and Check-Intention($S$, $d$, $i$) do {
            TransactionStart($S$);
            if precondition($S$, $d$) and impact($S$, $d$) then {
                TransactionCommit($S$);
                forall $tr$ such that RDF($d$, lrm:to, $tr$) \in $S$ and FWD-Change($S$, $i$, $r$, $M$)
                forall $tr$ such that RDF($d$, lrm:to, $tr$) \in $S$ do Set-Change($S$, $tr$)
            } else TransactionAbort($S$)
        }
    }
```

**Algorithm 1:** propagating the change of a resource

Note that this algorithm handles correctly the composition of dependencies, provided that the preconditions associated with the dependency node check for the disjunctive or conjunctive conditions among the source entities. In the example of Figure 4, a change of the “Parser 1” object will be propagated to the “comment stripper” object (typically, the associated impact would be building a new executable object) only if the precondition attached to the “Disjunctive Dependency” node defined that the “Parser 2” node changed too.

### 3.4 Context-aware Change Propagation for Targeted Resource (Local) Updating

The forward chaining algorithm presented in Section 3.3 propagates the changes to the whole dependency graph. In Algorithm 2, the idea is to examine a particular resource $r$ and search if it is impacted by one or several changes that may have occurred, exploiting the knowledge stored in the upstream
dependency graph. Cycles are handled through the marking mechanism used also in Algorithm 1.

```java
boolean BWD-Change(Ø, I, r, M) is
  // Ø is a store, I an Intention descriptor, r a resource
  // M is a set to memorize the exploration path
  var changed : Boolean = false
  if Check-Change(Ø, r) then { changed := true }
  else {
    if not (r ∈ M) then {
      set r ∈ M ; changed := false;
      TransactionStart (Ø);
    // d is a dependency
    for all d such that RDF(d, lrm:to, r) ∈ Ø and Check-Intentions(Ø, d, I) ∈ Ø do {
      // sr is a resource
      for all sr such that RDF(d, lrm:from, sr) ∈ Ø {
        if BWD-Change(Ø, I, sr, M) then changed := true
      }
    if changed then changed := precondition(Ø, d) and impact(Ø, d)
    if changed then TransactionCommit (Ø)
    else TransactionAbort (Ø)
    unset r ∈ M
  }
  return changed
```

Algorithm 2: examining whether a particular resource has been updated and accordingly propagating updates in the dependency graph

What makes the mechanism powerful, but more complex to handle, is the non-monotonicity of the change computation: local decisions taken during the exploration of the dependency graph may have a global effect, and therefore, may impact future precondition evaluations. Hence, there is a major difference with the forward chaining algorithm: actions associated with each dependency must be undertaken locally during the backtracking process (when preconditions are satisfied), but as they will not be necessarily retained at the end of the process, they must be reversible. To offer this important property, we rely on the nested transaction mechanism described in Section 3.2.

Note that a resource might be impacted by several dependencies; in that case, there is no predefined assumption about the consistency of the result (inconsistency may appear, for instance, if one action updates the target resource, while another action originating from another upstream graph deletes the resource). The coherence, if any should be ensured, must derive from the structural properties of the dependency graphs. The basic algorithm above applies all relevant actions (regarding the topology and preconditions), regardless of their global consistency or inconsistency (actually, we consider that the behavior of the system for such cases should be detailed at the level of the change management policy).

4. CASE STUDY

4.1 Case Study in Document Governance

We illustrate the advantages of our method over a method that does not take intent and context in general into account with the help of an example. We deal here with the simple task of having two different assemblies of a Visual Studio project, one with (part of) the comments stripped out/transformed and one with all of the comments included in the assembly. The first case is usually happening in a delivery/release setting (when delivering the software to the client we may want (part of) the comments stripped out), while the second in a development setting (the comments are available to the development team). When using Visual Studio, the comments to be rendered are included in corresponding similarly named XML documents. It is also frequent to have another setting where the comments delivered to a client have to follow a format/style that is defined by the policies and or practices of the specific client.

4.1.1 Change Cases

Taking into consideration the above requirements we have the following possible change cases.

Case 1. The code changes: Both release and development versions of the assembly have to take this into account.

Case 2. The XML document including the comments changes. In that case we have the following possibilities:

Case 2.1. We want the changes to be visible only to the development team and/or the comments included in the release version to not be impacted. In that case, no change takes place in the release mode, changes take place in the development mode only.

Case 2.2. We want the changes to be visible in the release mode as well. The comments included in the release version are then impacted and changes to both the release and development versions are required.

Case 3. The regulations of the company for comment formatting/styling change. In that case changes to the release version are required but not to the development version.

4.1.2 Participating Entities

It should be obvious from the above list that at least the following objects need to be represented in this case.

Object 1: The DLL files (Here to simplify we will consider only one DLL file). Henceforth noted DLL.

Object 2: The XML file holding the documentation for the DLL file. Henceforth noted XMLOrig.

Object 3: An XSLT file that holds the transformations that need to be executed to the original XMLOrig so that the resulting XML file that will be included in the final assembly is following the company regulations and/or defines which part of the comments should be stripped out/transformed. Henceforth noted XSLT.

Object 4: The resulting XML file after the transformation, noted XMLTrans.

Object 5: The rendered object R.

---

5 There are cases where this is an interesting feature, as some comments should not be viewed by the client.

6 XML documentation files are separate from the DLLs for an assembly. They are just UTF8 XML files. If they are in the same folder as the DLL that they document, and having the same name, then Visual Studio will recognize them and link them appropriately.
For completeness we also represent two different IDEs (VS1, VS2) that can be used to render the code and comments. We have to note here that for educational reasons we have greatly simplified this example leaving out a number of other objects that would influence the state of the above entities such as the hardware used.

4.1.3 Dependencies

The dependencies are related to the following three intentions: render, release, and develop (see also Figure 5 and Appendix A). For simplicity we do not include in the Figure the intention related to the transformation of the XML document XMLOrig (i.e. transform). Two main dependency types are represented in this example. One related to the generation of the rendering object and one to the transformation of the XML file (details in Appendix A).

Furthermore, the rendering can be sub-classified into rendering for the development team (rendDevDep in Figure 5), in which case the DLL and XMLOrig objects are necessary, and rendering for the client (rendRelDep in Figure 5), in which case the DLL and XMLTrans objects are required. The topology of the resulting dependency graphs is shown in Figure 5. Accordingly, a partial snapshot of the knowledge base holding the different entities at time t formalised in the Web Ontology Language (OWL 2) and serialised in Turtle [22] can be found at Appendix A.

4.1.4 Context-aware Change Propagation for the Change Cases

Case 1. Code changes imply DLL changes, so both release and development versions of the assembly have to take this into account. This is a standard case supported by existing systems and is not detailed further here.

Case 2. XML document changes imply XMLOrig changes.

Case 2.1. We do not want the changes to be propagated in the release mode and/or comments included in the release version are not impacted.

The above change is context-specific where the context is defined by the develop intention and the specific topology of the dependency graph. This means that although XMLOrig is modified, changes are not propagated to the XMLTrans entity, as the intention release is not compatible with the intention develop. This is defined in OWL as ex:Release owl:disjointWith ex:Develop and following from the fact that release is an instance of the ex:Release class while develop is an instance of the ex:Develop class (see Appendix A).

Please note that incompatibility here is defined in terms of OWL-based inconsistency identification. The corresponding dependency graph is shown in Figure 6.

Case 2.2. This is a standard case supported by existing systems and is not detailed further here.

Case 3. This implies that the XSLT object changes.

The above change is also context-specific as the change should only be propagated in the release mode. Contrary though to Case 2.1 the XSLT object is not shared between the two different intentions. However, such transformation makes sense only if XMLOrig is well-formed. If it is not well-formed no change propagation will take place. This constraint is formalised as part of the preconditions of the xmlTransDep dependency (see Appendix A).

4.2 Examples of Use in Other Domains

We have to note here that our method is applicable to different domains. For instance, within the context of the European project PERICLES, the same method will be used for the preservation of software and/or video-based digital art. [7], [10] present an example of a corresponding problem in that domain.

Another example is software versioning. In [11] a method for software versioning was presented, which encoded backward-compatibility in the version labels. We believe that this method could be extended by the integration of the dependency-based propagation method presented in this paper and therefore taking into consideration more general context including intent. It is well known that software written to perform the same functionality can be written in different ways. For instance, [12] describes two different ways of mixing bits from a pointer p1 with bits from a pointer p2, corresponding to different intents (Table 1 and Table 2).

Table 1. C code mixing some bits from a pointer p1 with bits from a pointer p2 with the intention of assigning either p1 or p2 to r in constant time

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * f2(int c, void * p1, void * p2)</td>
<td></td>
</tr>
<tr>
<td>void * r = p1;</td>
<td></td>
</tr>
<tr>
<td>uintptr_t mask = -c;</td>
<td></td>
</tr>
<tr>
<td>r = (void*)((uintptr_t)r &amp; mask)</td>
<td></td>
</tr>
<tr>
<td>(uintptr_t)p2 &amp; ~mask));</td>
<td></td>
</tr>
<tr>
<td>return r;</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. C code mixing some bits from a pointer p1 with bits from a pointer p2 with the intention of assigning either p1 or p2 to r as fast as possible without any guarantee on constant time computation

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * f1(int c, void * p1, void * p2)</td>
<td></td>
</tr>
<tr>
<td>void * r = p1;</td>
<td></td>
</tr>
<tr>
<td>if ((c) r = p2;</td>
<td></td>
</tr>
<tr>
<td>return r;</td>
<td></td>
</tr>
</tbody>
</table>

Nowadays, there is no available system that would allow us to keep both variants and version them while keeping explicit the intent-specific semantics into the versioning process. However, this can be a real need for professional software development (e.g. see [24] for a discussion on intents in software development).

5. RELATED WORK

There are two main streams of work on dependency-based representations for digital preservation. In the first one, dependencies are considered as simple binary relations while in the second one as complex constructs, although in both cases the notion of intention and in general context is not considered.

In the first stream the most important work includes the PREMIS Data Dictionary [14], which defines three types of relationships between objects: structural, derivation and dependency. Derivation and dependency relationships are the most relevant in our setting. A derivaton relation results from the replication or transformation of an object. A dependency relation exists when one object requires

---

7 Because of the limited space we put this content in Appendix A: http://download.xrce.xerox.com/doceng2016/doceng2016_Appendix.zip

8 www.pericles-project.eu/
Figure 5. Topology of the dependency graph at time t with three different intentions shown: render, release and develop. The dependency instances are xmlTransDep (transformation dependency), rendRelDep (rendering dependency for release version i.e. the client), rendDevDep (rendering dependency for development version i.e. the engineering team). Notice that the dependency paths can be different according to the intention, but also the same intention instances (and in general entities) can belong to different dependency paths (e.g. render, XMLOrig).

Figure 6. Topology of the dependency graph at time t''>t (except for the object R' that is kept to highlight the change that happened). Note that only the part of the dependency graph related to the develop intention is active for change propagation.
another to support its function, delivery, or coherence. Examples include a font, style sheet, DTD or schema that are not part of the file itself. Objects can also be related to events through user-defined dictionaries of terms, and events can in turn be linked to agents that performed those events.

Another widely used model is the Open Provenance Model (OPM) [15] that introduces the concept of a provenance graph, which captures causal dependencies between entities. The most relevant concept from our perspective is process that represents actions performed on or caused by artefacts, and resulting in new artefacts.

Interestingly also in hardware verification and design dependency relations are considered. The closest work is [19] where the authors focus on “design intent”, which generally refers to what the designer intends in terms of the interaction between components of the design and the designer’s expectations regarding acceptable functional behavior. [19] also refers to “implied design intent”. In any case this is not modelled as an aspect of the dependencies. Other works where dependency is considered a binary relation and intent is not taken into account include [20], [21].

In the second stream, the most interesting line of work, and most similar to ours, is from Tzitzikas and colleagues ([16], [17], [13]).

[16] defines the notions of module, dependency and profile to model use by a community of users. A module is defined as a software/hardware component or knowledge base that is to be preserved, and a profile is the set of modules that are assumed to be known. A dependency relation is then defined by the statement that module A depends on module B if A cannot function without B. For example, a .txt file depends on the availability of a text editor. The authors of [17] also define the more specific notion of task-based dependency, expressed as DataLog rules and facts. For instance, Compile(HelloWorld.java) denotes the task of compiling 'HelloWorld.java'. Since the compilation of the latter depends on the availability of a compiler, this dependency can be expressed using a rule of the form:

\[
\text{Compile}(X) :\text{ Compilable}(X,Y);
\]

where the binary predicate \text{Compilable}(X,Y) denotes the appropriateness of Y for compiling X. This formal approach enables various tasks to be performed, such as risk and gap analysis for specific tasks, possibly considering contextual information, such as user profiles.

In addition to the above work, [13] introduce the notion of \textit{intelligibility} (which seems to be related to the notion of \textit{task} defined in [17]), which allows for typing dependencies. Our approach goes one step further toward genericity, by allowing any kind of dependency specialization, the intelligibility being replaced by the notion of intention, which can be described informally or formally through additional properties. Moreover, LRM offers a much richer topology for dependency graphs through managing dependencies as instances instead of properties. By combining genericity and semantic refinement, we expect a tighter computational context (as in our case).

Change propagation and impact analysis have also been traditionally very important in software evolution. Most of the existing work, however, does not take into account contextualised dependencies and/or different intentions. One of the most interesting exceptions is the work presented in [18], where an agent-oriented approach is adopted to perform change propagation for the purpose of software evolution. In this case, the belief-desire-intention (BDI) model is followed, which is actually a way of separating and selecting a plan from the execution of currently active plans. Intentions represent the deliberative state of the agent – what the agent has chosen to do which relates to the execution of a plan. Of course the setting and overall setup is different than ours, but also most importantly, they do not define explicit representation structures for representing the dependencies between different entities. They rather assume that they have a set of plans which have specific goals and the only connection between the entities is encoded within those plans. The context then is defined as an additional or modified goal that can change during execution of the plan. Constraints on plans are encoded as rules. In addition, no change propagation algorithms guaranteeing local computation and dealing with transactions, as in our case, is proposed.

6. CONCLUSION

In this paper we have presented a method that allows taking into account initial development and change intentions, as well as relevant context, for document and data governance. To achieve that we view the collection of documents or in general digital entities to be preserved and managed, as a network of entities where the dependencies between them are not simple binary links but complex, semantically rich, constructs that encompass context-related information. Context includes conditions under which change can have an impact on (part of) the network, characterizing that impact, and linking it to change and development intents. Furthermore, our model is extensible, allowing specializing the dependencies and adding more properties if required.

Intent is explicitly represented as a first class citizen in our representation. This enables to parameterize dependency interpretation: change interpretation becomes a potentially complex function of intents and relations between them (e.g. by (re)using a specialised ontology). This allows broad inferences about class membership and classification, or inconsistency identification (as illustrated in our case study) and resolution based on specific context interpretations.

An important aspect of the algorithmic part of our method is that we allow changes that may result in undesirable side-effects e.g. operations that have not successfully terminated can change the state of the actual dependency network in an incorrect way. This is possible thanks to the introduction of a nested transactional mechanism. The same mechanism could be used to enable simulations (e.g. if I change that object, what is the effect on the rest of the objects?), which is an essential aspect of risk analysis.

As illustrated through our case study the advantages of our method when compared to existing ones include: more fine grained change impact calculation and analysis; genericity, as heterogeneous digital objects can be preserved in different contexts; and incompatibility detection via ontology-based reasoning, by specializing/configuring dependencies.

In the near future we would like to extend the inference capabilities presented in this paper with the use of a change impact analysis infrastructure that will allow even more expressivity in the notion of preconditions and impacts. We would also like to apply our methodology in software change management, where intention recording and risk analysis in the face of change are very important.
7. ACKNOWLEDGMENTS
This project has received funding from the European Union’s Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 601138

8. REFERENCES