Dependency Management for the Preservation of Digital Information

Yannis Tzitzikas

Computer Science Department, University of Crete, GREECE, and
Institute of Computer Science, FORTH-ICS, GREECE
tzitzik@ics.forth.gr

Abstract. The notion of dependency is ubiquitous. This paper approaches this notion from the perspective of digital information preservation. At first, an abstract notion of *module* and *dependency* is introduced. Subsequently, and for building preservation information systems, the notion of *profile* is proposed as a gnomon for deciding *representation information adequacy* (during input) and *intelligibility* (during output). Subsequently some general dependency management services for identifying and filling gaps during input and output are described and analyzed (also described as protocols that could be used in the communication between a preservation information system and information consumers and providers).

1 Introduction

The preservation of digital information is an important requirement of the modern society. Digital information has to be preserved not only against hardware and software technology changes, but also against changes in the knowledge of the community. According to the OAIS reference model [2], metadata are distinguished to various broad categories. One very important (for preservation purposes) category of metadata is named *Representation Information* (RI) which aims at enabling the conversion of a collection of bits to something useful. In brief, the RI of a digital object should comprise information about the Structure, the Semantics and the needed Algorithms for interpreting and managing a digital object. Figure 1 shows one corresponding part of the information model of OAIS.

In order to abstract from the various domain-specific and time-varying details, in this paper we model the RI requirements as *dependencies*. This view is very general and can capture a plethora of cases. Subsequently, we identify a set of core services for managing dependencies. These services aim at identifying the knowledge gaps (missing RI), and at computing and proposing ways to fill these gaps. These services can be used during both importing and exporting information (to and from a preservation information system). As different users (consumers or providers), or communities of users, have different characteristics (in terms of RI), we introduce the notion of DC (Designated Community) profile. Subsequently, we describe protocols (interaction schemes) that could be

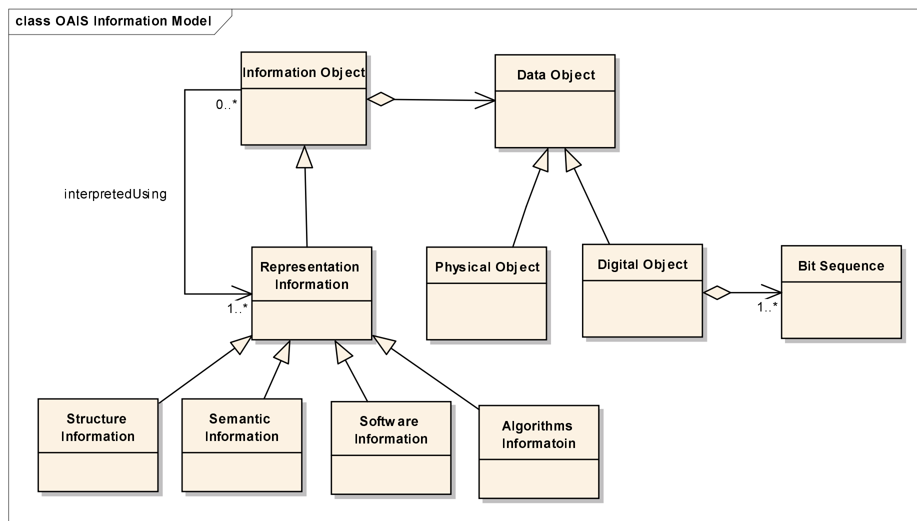

Fig. 1. The information model of OAIS

used in the communication between a preservation information system and information consumers and providers. This work can be exploited for building advanced preservation information systems and registries. Motivation for this work is the ongoing EU project CASPAR (FP6-2005-IST-033572) whose objective is to build a pioneering framework to support the end-to-end preservation lifecycle for scientific, artistic and cultural information.

The paper is organized as follows. Section 2 formalizes the notion of dependency and knowledge gap, while Section 3 describes interaction schemes for identifying and filling these gaps. Finally, Section 4 concludes the paper and identifies issues for further research.

2 Formalizing Dependencies

Let $Obj = \{o_1, \ldots o_n\}$ be set of all objects of the domain, e.g. the set of all data objects of an archive. Let $\mathcal{T}$ be the set of all *modules* (or components) that are needed for understanding/executing/managing the objects in Obj. We adopt a very general interpretation of the term module. It can be a software or hardware module. In addition, it could be a knowledge model expressed either formally or informally, explicitly or tacitly. For instance, it could be an English-To-Greek dictionary that is useful for a Greek-speaking person to understand a piece of text written in English. It could also be a ontology A (which could be expressed in RDF/S) that is useful for understanding the contents of a metadata file (expressed in RDF), or for understanding another ontology B (e.g. if B uses or specializes elements defined in A).

There is dependency relation between modules in the sense that a module may require the availability of one or more other modules in order to function. We can model this as a graph $\Gamma = (\mathcal{T}, <)$. A relationship $t < t'$ means that t'

depends on t, e.g. it may mean that t' cannot function without the existence of t. Below we describe some small examples (based on the needs of the CASPAR project). Figure 2(a) shows the dependencies of a text file written in English. However, a Greek-speaking consumer may define a dependency graph like the one illustrated in Figure 2(b).

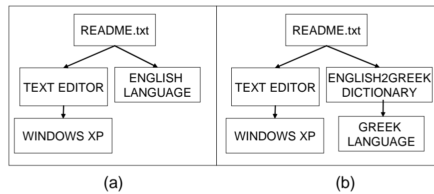

Fig. 2. Dependencies for a text file

FITS[1] is a standard data format that is used in astronomy. To understand such a file one needs to understand the FITS standard which is in turn described in a PDF document. To understand the keywords contained in a FITS file one needs to be able to understand the FITS dictionary (that explains the usage of keywords). Figure 3(a) illustrates these dependencies, while Figure 3(b) shows the dependencies of a digital object representing an interactive multimedia performance. Finally, an example of dependencies between formal knowledge expressed in the form of RDF Schemas is shown in Figure 4 (where fat arrows are used to denote dependencies between namespaces).

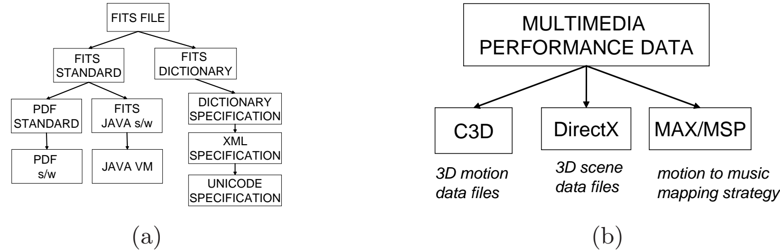

Fig. 3. Dependencies of scientific and multimedia performance data

A general remark is that there is no standard method for defining what a module is. For instance, we may have modules of various levels of abstraction. One module in one dependency graph could correspond to a large number of interconnected and interdependent finer modules in another dependency graph. For instance, the `WINDOWS XP` module in Figure 2 is actually the aggregation of several interconnected modules[2]. Hereafter we shall make the working assumption

[1] http://fits.gsfc.nasa.gov/

[2] Hierarchical clustered graphs could be probably used for modeling and formalizing the dependencies among modules of different granularity, but this goes beyond the scope of this paper.

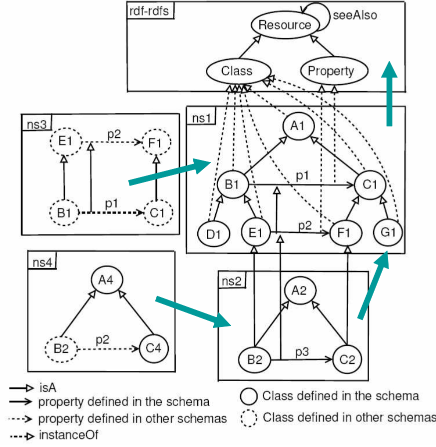

Fig. 4. Dependencies between RDF Schemas

that the dependency graph is acyclic, i.e. it is a DAG. Equivalent modules (e.g. all editors that can read and edit ASCII texts) can be captured by assuming that each element of $\mathcal{T}$ is not atomic but it is a set of equivalent modules (this is like having disjunctive dependencies).

The intelligibility of a digital object, i.e. of an element in Obj, may require the existence of one or more modules in $\mathcal{T}$. We can model this by a binary relation R ($R \subseteq Obj \times \mathcal{T}$). To keep notations simple we abuse notation and we will also use $<$ to denote R. So we can view all dependencies (among modules and between object and modules) as one graph $\Gamma = (T \cup Obj, <)$. For example, if the management of an object o requires two modules t_1, t_2 where t_2 requires a module t_3 we can write $t_1 < o$, $t_2 < o$, $t_3 < t_2$. Table 1 introduces some notations that will be used in the sequel.

Table 1. Notations

Notation	Definition
$\mathcal{T}$	the set of all modules and objects
t	an element of $\mathcal{T}$
S	a subset of $\mathcal{T}$
$t < t'$	t' depends on t (in other words, t' requires t)
$<^*$	the transitive closure of $<$
$min(S)$	the minimal elements of S w.r.t. $<^*$
$max(S)$	the maximal elements of S w.r.t. $<^*$
$Nr(t)$	$\{ t' \mid t' <^* t\}$, i.e. all modules that t requires
$Nr(S)$	$\cup\{ Nr(t) \mid t \in S\}$

The minimal elements of $\mathcal{T}$, i.e. the set $min(\mathcal{T})$, comprises the primitive modules which are assumed to be always available (e.g. an Operating System, a programming language, or the English vocabulary). However, probably nothing

in this world is self-existent so the notion of primitive modules is actually a convention.

Regarding OAIS, we could say that the `interpretedUsing` relation of Figure 1 defines a plain dependency graph with the only difference that the nodes of this graph may be further specialized, i.e. classified under the indicative categories that are shown (e.g. `Algorithm, Semantics, Structure`, etc). In any case, the resulting object graphs would contain a dependency graph like the one we have introduced so far.

2.1 Intelligibility of Data Objects

Given an object or module t, we can define the required for understandability (or intelligibility) modules of t as follows: $req(t) = Nr(t)$. If $S \subseteq \mathcal{T}$, then we can define $req(S) = \cup\{ req(t) \mid t \in S\}$. Let u be an actor (e.g. user, or information consumer) and let T_u be the modules available to him (e.g. software/hardware modules available at his computer or knowledge available at his/her mind), where $T_u \subseteq \mathcal{T}$. Now suppose that u is given a set of objects A ($A \subseteq Obj$). The set A could be the answer of a query q posed to an information system, or the result of browsing an information space, or the result of any other method (e.g. u may have received the set A by email). The prerequisites for understanding the set A is $req(A)$. For example, consider the case illustrated in Figure 5 where $\mathcal{T} = \{t_1, \ldots, t_8\}$, $A = \{o_x, o_y\}$, and $T_u = \{t_3, t_6\}$. Since T_u contains t_6 and none of its narrower modules t_7 and t_8, we can understand that t_6 is a primitive module for u. So we can safely make the assumption that u knows t_7 and t_8. We can call this *unique module assumption (uma)*, meaning that each module is uniquely identified by its name and that its required modules are always the same. Here we have $req(o_x) = \mathcal{T}$ and $req(o_y) = \{t_3, t_6, t_7, t_8\}$. Also note that $max(req(o_x)) = t_1$ and $max(req(o_y)) = t_3$.

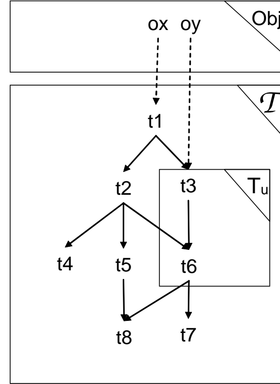

Fig. 5. Example of a dependency graph (between objects and modules)

We can easily see that u can understand an object o if $max(req(o)) \subseteq T_u$. In the current example u can understand o_y because $max(req(o_y)) = t_3 \in T_u$, however u cannot understand o_x because $max(req(o_x)) = t_1 \notin T_u$.

2.2 Intelligibility Gaps

Consider the case of an object o_x that is not understandable by u. In this case we can say that we have an intelligibility gap. To fill the gap, we need to find the missing modules. The set of missing modules that u needs in order to understand an object o are given by the formula: $Missing(o,u) = req(o) - Nr(T_u)$. In our example, $Missing(o_x,u) = req(o_x) - Nr(T_u) = \mathcal{T} - \{t_3,t_6,t_7,t_8\} = \{t_1,t_2,t_4,t_5\}$. Clearly, if $A \subseteq Obj$, then we can define $Missing(A,u) = \cup\{\ Missing(o,u) \mid o \in A\}$.

Note that without the unique module assumption (uma), we could not make the assumption that u knows t_7 and t_8. In that case we would have to define $Missing(o,u) = req(o) - T_u$. In our running example we would have $Missing(o_x,u) = req(o_x) - T_u = \mathcal{T} - \{t_3,t_6\} = \{t_1,t_2,t_4,t_5,t_7,t_8\}$. The relationships between two dependency graphs are specified formally below.

Consider an information provider p and an information consumer u, each having a dependency graph Γ_p and Γ_u respectively.

Definition 1. Let $\Gamma_u = (T_u, <_u)$ and $\Gamma_p = (T_p, <_p)$ be two dependency graphs. We say that Γ_u is *subgraph* of Γ_p, and we write $\Gamma_u \subseteq \Gamma_p$, if (a) $T_u \subseteq T_p$, and either (b1) $<_u \subseteq <_p$, or (b2) $<_u = <_{p|T_u}$.

Note that (b2) is more strict than (b1). Specifically, and regarding the relationships between the elements of T_u, (b1) ensures that p has at least the relationships that u has, while (b2) ensures that p has exactly the relationships that u has. For instance, Γ_u of Figure 5 satisfies (b2). If Γ_u did not contain the relationship $t_6 < t_3$ then it would satisfy only (b1). If Γ did not contain the relationship $t_6 < t_3$ then it would not satisfy neither (b1) nor (b2). Note that: (b1) implies that $Nr_p(t) \supseteq Nr_u(t),\ \forall t \in T_u$, while (b2) implies that $Nr_p(t) = Nr_u(t),\ \forall t \in T_u$.

3 (Intelligibility-aware) Interaction Schemes

Consider an information provider p and an information consumer u. Here we describe various interaction methods that could be used for identifying and filling intelligibility gaps.

3.1 For Consuming (Delivering) Information

Without loss of generality we can assume a *query-and-answer* interaction scheme where u sends to p a query q and p returns a set of objects A. Below we describe some interaction schemes that enrich the query-and-answer interaction scheme with intelligibility-related concerns.

Note that given an object o and a user u, for computing $Missing(o,u) = req(o) - Nr(T_u)$ one needs to be able to compute $req(o)$ and $Nr(T_u)$. If $req(o)$ or $Nr(T_u)$ are very large in size then this could cause inefficiencies (especially in a distributed setting). For this reason below we describe a number of options.

Interaction Schemes with Fixed Number of Messages

(A) u submits a query, p returns the answer with all modules that are required.
(1) $u \rightarrow p$: query(q)
(2) $p \rightarrow u$: return$(A, req(A))$
Note that $req(A)$ does not necessarily return the modules themselves. It may return references to these modules which one could use in order to find the actual modules (e.g. for downloading and installing them). The user can identify the missing modules (i.e. those elements in $req(A)$ which are unknown to him) and proceed accordingly. However in practice $req(A)$ could be very large in size.

(B) u submits her query and profile, p returns answers accompanied by the missing modules.
(1) $u \rightarrow p$: query(q, T_u)
(2) $p \rightarrow u$: return$(A, Missing(A, u))$
Note that if T_u is smaller than $req(A)$ then this scheme is more efficient than (A). We can further improve the above scheme, specifically we can reduce the data that have to be exchanged, if $\Gamma_u \subseteq \Gamma_p$. In particular, in that case step (1) can be replaced by:
(1') $u \rightarrow p$: query$(q, max(T_u))$

(C) u registers her profile once, p returns answers accompanied by the missing modules. This scheme avoids sending the profile with each query. Instead, u registers a (DC) profile T_u once, which is then exploited in the subsequent query-and-answer interactions. Again the provider sends back the answer and the missing modules.
(1) $u \rightarrow p$: register(u, T_u)
(2) $u \rightarrow p$: query(q)
(3) $p \rightarrow u$: return$(A, Missing(A, u))$
We can further improve the above scheme, specifically we can reduce the data that have to be exchanged for the registration, if $\Gamma_u \subseteq \Gamma_p$. In particular, in that case, step (1) can be replaced by:
(1') $u \rightarrow p$: register$(u, max(T_u))$

Progressive Interaction Schemes (with variable number of messages)
In some cases it might be useful (or efficient) to provide gradual/progressive methods for identifying and filling intelligibility gaps. Two such schemes are described bellow.

(Ai) This is a progressive version of scheme (A). Instead of sending $req(A)$, the provider at first sends only the maximal elements.
(1) $u \rightarrow p$: query(q)
(2) $p \rightarrow u$: return$(A, max(req(A)))$
The user can identify the missing modules (i.e. those elements in $max(req(A))$ which are unknown to her) and proceed accordingly. Note that u could also ask again p about the required modules of the elements of $max(req(A))$ and so on, i.e. the dialog could be continued as shown next. Below we use $recmsg$ to denote the previously received message.

(3) u: repeat
(4) u: $M := recmsg - T_u$ // i.e. $M := max(req(A)) - T_u$
(5) u: If $M \neq \emptyset$ then
(6) $u \rightarrow p$: $getDirectReqsOf(M)$
(7) $p \rightarrow u$: return($max(req(recmsg))$)
(8) u: until $M = \emptyset$

For instance, in our running example the formula $max(req(o_x)) - T_u$ returns the highest missing module, i.e. t_1. The entire sequence of M's is shown below:
M_1: t_1 ($= max(req(o_x)) - T_u$)
M_2: t_2 ($= max(req(t_1)) - T_u$)
M_3: $\{t_4, t_5\}$ ($= max(req(t_2)) - T_u$)
M_4: $\{t_8\}$ ($= max(req(t_5)) - T_u$)
M_5: $\emptyset$
Note that t_8 could be already known to u as it is narrower than $t_6 \in T_u$.

(D) u submits only the query, p returns only the answer.
Here u sends to p only q. If u cannot understand the result, she can send to p what she did not understand. With the assumption that each object has links to its direct required modules, u can identify the direct missing modules and send these to p and continue in this way until reaching to her primitive elements or getting all elements of $req(A)$.

(1) $u \rightarrow p$: query(q)
(2) $p \rightarrow u$: return(A)
(3) u: repeat
(4) u: $M := computeDirectReqsOf(recmsg) - T_u$
(5) u: If $M \neq \emptyset$ then
(6) $u \rightarrow p$: $getDirectReqsOf(M)$
(7) $p \rightarrow u$: return($max(req(recmsg))$)
(8) u: until $M = \emptyset$

3.2 For Providing (ingesting) Information

A preservation system could follow a policy of the form: the dependencies of the stored objects should be known and stored. This means that the submission of information, e.g. the submission of an object or module t, to the system should be accompanied by adequate representation information. In other words $req(t)$ should be known. However as there is not any objective method for deciding whether $req(t)$ is complete or not (may nothing is complete in the strict sense) we can again use the notion of profile in order to decide whether the submitted RI is complete or not (with respect to a specific profile or with respect to all profiles known by the preservation system).

As one can imagine, the provision (ingestion) of information has many similarities with the consumption (delivery) of information. We could capture the ingestion of information by changing the previously described interaction schemes.

Specifically we could ignore the query submission step and consider that the user u is the preservation system who wants to ingest the set of objects A that p sends to u. Fore reasons of space their detailed description is omitted.

3.3 Complex Objects and Other Technicalities

Let us for example consider the case of Web pages. Consider a digital file named *a.html*. The extension of the filename gives us a hint about the type of the digital object, so we may write $type(a.html) = HTML$, and as $a.html > HTML$, we may generalize and consider that for every $o \in Obj$, it holds $o > type(o)$, if $type(o)$ is known. However, an html page is a text that may contain pointers to other types of data (images, sounds, etc). In order to obtain this content, we need a HTML parser. So we could say that $computeDirectReqsOf(a.html)$ needs the availability of an HTML parser[3]. Consequently, $computeDirectReqsOf(o)$ could be as follows: $computeDirectReqsOf(o) = type(o) \cup type(o).parse(o).getContents()$. To compute all required modules of an object we have to continue analogously.

4 Concluding Remarks

Dependencies are ubiquitous and dependency management is an important requirement that is subject of research in several (old and new emerged) areas, from software engineering [6–8, 1] to ontology engineering [3, 5]. In software engineering the various build tools (e.g. make, gnumake, nmake, jam, ant) are definitely related, as well as the problems of installability, deinstallability and maintainability. Recall that the art of large-scale design is to minimize dependencies (recall Model Driven Architecture). However we could say that the preservation of the intelligibility of digital objects requires a generalization (or abstraction) able to capture also non software modules (e.g. explicit or implicit domain knowledge). The agenda of ontology engineering includes similar in spirit problems, e.g. the problem of how to reflect a change of an ontology to the dependent ontologies (i.e. to those that reuse or extend parts of it), which may be stored in different sites, as well as the schema evolution problem, i.e. the problem of reflecting schema changes to the underlying instances.

A modern preservation system should be generic, i.e. able to preserve heterogeneous digital objects which may have different interpretation of the notion of dependency. The dependency relations should be specializable and configurable (e.g. it should be possible to associate different semantics to them). Focus should be given on finding, recording and curating the dependencies. For example, the `makefile` of an application program is not complete for preservation purposes. The preservation system should also describe the environment in which the application program (and the make file) will run. Recall the four worlds of an information system (Subject World, System World, Usage World, Development

[3] As another example, for a .java named file we need to parse the file in order to extract all import statements, while for a .rdf named file, we need to parse it in order to extract the namespaces it uses.

World) as identified by Mylopoulos [4]. Finally, the provision of notification services for risks of loosing information (e.g. obsolescence detection services) is important.

The contribution of this paper lies in specifying a generic view by adopting an abstract notion of module and dependency and by introducing the notion of DC profile. Subsequently it specified a number of core services around these notions, allowing to check and control whether the ingestion of information is complete and for computing the minimum extra information required to be delivered to ensure the intelligibility of a digital object by the consumer. Based on these services a number of interaction schemes for identifying and filling the intelligibility gaps were presented. A proof-of-concept prototype based on Semantic Web technologies has already been built. The benefits of adopting Semantic Web languages, for the problem at hand, is that although the core dependency management services need to know only a very small core ontology (defining the abstract notion of module and dependency), it is possible to refine (specialize) the dependency relation.

Issues for further research include (a) extending the framework with converters (for tackling migration/emulation), (b) studying the effects of changes in the dependency graphs (and what kind of notification services are required), and (c) studying composite modules and dependencies of different granularity.

Acknowledgements This work was partially supported by the EU project CASPAR (FP6-2005-IST-033572). Many thanks to David Giaretta and the rest "CASPARtners".

References

1. X. Franch and N.A.M. Maiden. Modeling Component Dependencies to Inform their Selection. *2nd Intern. Conf. on COTS-Based Software Systems*, 2003.
2. International Organization For Standardization. "OAIS: Open Archival Information System – Reference Model", 2003. Ref. No ISO 14721:2003.
3. M. Jarrar and R. Meersman. Formal Ontology Engineering in the DOGMA Approach. *International Conference on Ontologies, Databases and Applications of Semantics (ODBase)*, pages 1238–1254, 2002.
4. J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. "Telos: Representing Knowledge about Information Systems". *ACM Transactions on Information Systems*, 8(4), October 1990.
5. E. Sunagawa, K. Kozaki, Y. Kitamura, and R. Mizoguchi. An Environment for Distributed Ontology Development Based on Dependency Management. *Proc. of the 2nd International Semantic Web Conference (ISWC2003)*, pages 453–468, 2003.
6. M. Vieira, M. Dias, and D.J. Richardson. Describing Dependencies in Component Access Points. *Proceedings of The 23rd International Conference on Software Engineering (ICSE'01), Toronto, Canada*, pages 115–118, 2001.
7. M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. *ASE*, 00:241, 2002.
8. M. Walter, C. Trinitis, and W. Karl. "OpenSESAME: An Intuitive Dependability Modeling Environment Supporting Inter-component Dependencies". *Procs of 2001 Pacific Rim International Symposium on Dependable Computing*, pages 76–83, 2001.