University of Crete

Computer Science Department

Conceptual Modeling and Tools for Digital Preservation

Yannis Marketakis

Master's Thesis

Heraklion, April, 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Μοντέλα και Εργαλεία για την Διαφύλαξη Ψηφιακών Αντικειμένων

Εργασία που υποβλήθηκε απο τον

Γιάννη Μαρκετάκη

ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Γιάννης Μαρκετάκης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Γιάννης Τζίτζικας, Επίκουρος Καθηγητής, Επόπτης

Βασίλης Χριστοφίδης, Καθηγητής, Μέλος

Γρηγόρης Αντωνίου, Καθηγητής, Μέλος

Δεκτή:

Πάνος Τραχανιάς, Καθηγητής

Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Απρίλιος 2010

Conceptual Modeling and Tools for Digital Preservation

Yannis Marketakis
Master's Thesis
Computer Science Department, University of Crete

Abstract

Modern society and economy is increasingly dependent on a deluge of only digitally available information, therefore its preservation within an unstable and rapidly evolving technological (and social) environment is a challenging problem of prominent importance. The contributions of this thesis revolve around three main topics:

(a) Intelligibility. As it is hard to define explicitly what information or what knowledge is, it is therefore very difficult to claim that a particular approach, methodology or technique can indeed preserve information and knowledge. To tackle this issue and for preserving the meaning of digital objects, we formalized the notion of intelligibility in a OAIS-compliant manner and provided guidelines, methodologies and components that can aid humans in preserving information and knowledge. Specifically we formalized the notion of intelligibility and intelligibility gap through the notion of dependency. This perspective allows answering questions of the form: (a) what kind of (and how much) representation information do we need, (b) how this depends on the designated community, (c) what kind of automation could we offer (regarding packaging and dissemination). Apart from developing formal and conceptual models, we developed RDF/S ontologies and tools (`GapManager`) and applied them in real data in the context of the CASPAR project.

(b) Provenance Modeling and Querying. There is a need for a comprehensive and extensible conceptual framework to integrate, exchange and exploit provenance information within or across digital archives. We extended the ISO standard CIDOC CRM, defining CIDOC CRM Digital, to explicitly model digital objects and showed how it can be employed for expressing and querying provenance information.

(c) Automating the Ingestion and Transformation of Metadata. Most of the preservation approaches rely on metadata. However the creation and maintenance of metadata is a laborious task that does not always pay off immediately. There is a need for tools that automate as much as possible the creation and curation of preservation metadata. We developed `PreScan`, a tool for automating the ingestion phase. It can bind together automatically extracted embedded

metadata with manually provided metadata. It also supports processes for ensuring the freshness of the metadata repository and transforms metadata according to CIDOC CRM Digital.

Supervisor: Yannis Tzitzikas
Assistant Professor

Μοντέλα και Εργαλεία για την Διαφύλαξη Ψηφιακών Αντικειμένων

Γιάννης Μαρκετάκης
Μεταπτυχιακή Εργασία
Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Η σύγχρονη κοινωνία και οικονομία εξαρτάται ολοένα και περισσότερο από πληροφορίες εκφρασμένες αποκλειστικά σε ψηφιακή μορφή, επομένως η διατήρησή τους σε ένα ασταθές και συνεχώς μεταβαλλόμενο τεχνολογικό και κοινωνικό περιβάλλον είναι μία πρόκληση θεμελιώδους σημασίας. Η συμβολή αυτής της μεταπτυχιακής εργασίας στο πρόβλημα της ψηφιακής διαφύλαξης περιστρέφεται γύρω από τρία κεντρικά θέματα:

(α) Κατανοησιμότητα. Δεν αρκεί η διαφύλαξη των bits των ψηφιακών αντικειμένων, θέλουμε να διαφυλάξουμε και τη σημασία αυτών των bits. Καθώς είναι δύσκολο να ορίσουμε επακριβώς τι είναι πληροφορία ή γνώση, είναι επίσης πολύ δύσκολο να ισχυριστούμε ότι μία συγκεκριμένη προσέγγιση, μεθοδολογία η τεχνική μπορεί διατηρήσει την ψηφιακά εκφρασμένη πληροφορία ή γνώση. Για να αντιμετωπίσουμε αυτό το πρόβλημα προτείναμε τη μοντελοποίηση της έννοιας της κατανοησιμότητας με τρόπο συμβατό με το πρότυπο OAIS (Open Archival Information Systems, ISO 14721:2003) και εν συνεχεία ορίσαμε μία μεθοδολογία και συστατικά/εργαλεία λογισμικού για την πραγμάτωση αυτής της προσέγγισης. Συγκεκριμένα μοντελοποιήσαμε τις έννοιες της κατανοησιμότητας (intelligibility) και του κενού κατανοησιμότητας (intelligibility gap) μέσω της έννοιας της εξάρτησης. Αυτή η οπτική κάνει εφικτή την απάντηση ερωτημάτων όπως: (α) τι είδος (καθώς και πόση) πληροφορία αναπαράστασης (Representation Information) χρειαζόμαστε, (β) πως αυτή εξαρτάται από την κοινότητα για την οποία θέλουμε να διαφυλάξουμε την πληροφορία, (γ) τι είδους αυτοματισμούς μπορούμε να προσφέρουμε (όσον αφορά την αρχειοθέτηση και τη διανομή ψηφιακών αντικειμένων). Εκτός από την ανάπτυξη τυπικών μοντέλων, ορίσαμε τα μοντέλα σε RDF/S, και αναπτύξαμε εργαλεία και εφαρμογές (`GapManager`), και τέλος τα χρησιμοποιήσαμε σε πραγματικά δεδομένα στα πλαίσια του ευρωπαϊκού έργου CASPAR.

(β) Μοντελοποίηση και Επερώτηση πληροφοριών Προέλευσης. Υπάρχει ανάγκη για ένα πλούσιο και επεκτάσιμο εννοιολογικό πλαίσιο για την ενοποίηση, ανταλλαγή και διαχείριση πληροφοριών που αφορούν στην προέλευση των ψηφιακών αντικειμένων. Για το σκοπό αυτό επεκτείναμε το ISO πρότυπο CIDOC CRM, και ορίσαμε το CIDOC CRM Digital, ώστε να καλύπτει ψηφιακά αντικείμενα και δείξαμε πως μπορεί να χρησιμοποιηθεί για την παράσταση και επερώτηση πληροφοριών

προέλευσης.

(γ) Αυτοματοποίηση της Λήψης και Μετασχηματισμού των Μεταδεδομένων. Οι περισσότερες προσεγγίσεις που αφορούν στην ψηφιακή διατήρηση βασίζονται σε μεταδεδομένα. Ωστόσο η δημιουργία και η συντήρησή τους είναι μία κοπιαστική διαδικασία που δεν αποδίδει άμεσα οφέλη (αν τα μεταδεδομένα αυτά δεν εξυπηρετούν επιχειρησιακούς σκοπούς αλλά συλλέγονται μόνο για λόγους μακρόχρονης διαφύλαξης). Ως εκ τούτου υπάρχει η ανάγκη για εργαλεία τα οποία αυτοματοποιούν τη δημιουργία και επιμέλεια των μεταδεδομένων διατήρησης. Για το σκοπό αυτό σχεδιάσαμε και αναπτύξαμε το `PreScan`, ένα εργαλείο για την αυτοματοποίηση της διαδικασίας συλλογής μεταδεδομένων. Συγκεκριμένα το `PreScan` σαρώνει συστήματα αρχείων και από κάθε αρχείο εξάγει τα ενσωματωμένα μεταδεδομένα και τα μετασχηματίζει σε οντολογικά (εκφρασμένα βάσει της οντολογίας CIDOC CRM Digital). Επίσης επιτρέπει τον εμπλουτισμό τους από το χρήστη και προσφέρει μηχανισμούς για την ανανέωση του αποθετηρίου κατόπιν αλλαγών στο σύστημα αρχείων.

Επόπτης Καθηγητής: Γιάννης Τζίτζικας
Επίκουρος Καθηγητής

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επόπτη καθηγητή μου κ. Γιάννη Τζίτζικα για την πολύτιμη καθοδήγηση και ουσιαστική συμβολή του για την ολοκλήρωση αυτής της εργασίας. Επίσης τον ευχαριστώ για την εμπιστοσύνη που μου έδειξε κατα την διάρκεια της συνεργασίας μας.

Επίσης θα ήθελα να ευχαριστήσω τον καθηγητή κ. Βασίλη Χριστοφίδη για τις εύστοχες παρατηρήσεις του και συμβουλές για την περάτωση αυτής της εργασίας, καθώς επίσης και για την συμμετοχή του στην τριμελή εξεταστική επιτροπή της μεταπτυχιακής μου εργασίας.

Επιπρόσθετα θα ήθελα να ευχαριστήσω και τον καθηγητή κ. Γρηγόρη Αντωνίου για την προθυμία του να συμμετάσχει στην τριμελή εξεταστική επιτροπή της μεταπτυχιακής μου εργασίας.

Παράλληλα θα ήθελα να ευχαριστήσω το Εργαστήριο Πληροφοριακών Συστημάτων του Ινστιτούτου Τεχνολογίας και Έρευνας. Ειδικότερα θα ήθελα να ευχαριστήσω θερμά τους απόφοιτους του Τμήματος Επιστήμης Υπολογιστών Τηλέμαχο Τζανάκη και Δανάη-Χριστίνα Κομητοπούλου για την ενασχόληση τους με κομμάτια αυτής της εργασίας.

Ένα μεγάλο ευχαριστώ θα ήθελα να δώσω σε όλους τους φίλους μου για την υποστήριξη. Θα ήθελα να ευχαριστήσω ξεχωριστά τους Κατερίνα Δεμενεοπούλου, Γιάννη Σπερελάκη και Μπάμπη Τζαγκαράκη για την ηθική συμπαράσταση τους σε δύσκολες και εύκολες στιγμές, την κατανόηση τους και τα όσα περάσαμε μαζί.

Τέλος θα ήθελα να εκφράσω την ευγνωμοσύνη μου στους γονείς μου Γιώργο και Μαρία και την αδερφή μου Ειρήνη για την στήριξη, την ενθάρρυνση και την ολόψυχη αγάπη τους σε κάθε βήμα της ζωής μου. Σας ευχαριστώ πολύ για όλα.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Until a few decades ago information existed only in physical form. Information was written or engraved on several materials such as stone, wood, papyrus, silk, paper and was human recognizable. The preservation of information stored in physical materials was not a problem. The only action that should be followed was to store these materials in a safe place in order to protect them from destruction, either from natural disaster (flood, fire, earthquake) or corrosion of materials. However the nature of these materials made difficult the replication and distribution of the information.

Then the coming of digital ages provided a solution to these problems. Information obtained a digital representation, new documents were created and disseminated to different consumers within just a few minutes. Additionally digital objects could be easily replicated without loss or degradation. Documents were created and disseminated in digital form and objects that were not born-digital were transformed into digital ones. Furthermore the digital documents covered only a little space compared to their analog counterparts, hence large collections of documents could now fit in a few CDs.

However digital objects need specific viewer applications to be interpreted. These in turn depend on specific libraries, operating systems and hardware devices. Furthermore digital information is stored in several storage media ranging from magnetic tapes, magnetic and optical disks to volatile storage (RAM). Consequently digital objects became extremely vulnerable to the software that is used to interpret it and the hardware that is used to store it, since if any of the layers of the dependency tree is lost, the object will be inaccessible and useless. Additionally there are vulnerabilities regarding the interpretation of digital objects and the documentation of their provenance. So there is an increasing need to preserve digital objects, in essence to ensure

that digital objects will remain accessible and usable in future.

1.1 What is Digital Preservation?

Digital Preservation refers to the series of managed activities necessary to ensure continued access to digital materials for as long as necessary. It refers to all of the actions required to maintain access to digital materials beyond the limits of media failure or technological change. Those material may be records that were born-digital or are the products of a digitization process. In brief we could identify the following digital preservation approaches

- **Long-term preservation** - Continued access to digital materials, or at least to the information contained in them, indefinitely.
- **Medium-term preservation** - Continued access to digital materials beyond changes in technology for a defined period of time but not indefinitely.
- **Short-term preservation** - Access to digital materials either for a defined period of time while use is predicted but which does not extend beyond the foreseeable future and/or until it becomes inaccessible because of changes in technology.

The preservation of digital objects entails far more than making backup copies and storing them in disparate locations. Digital preservation is a series of managed activities necessary for ensuring both the long-term maintenance of the digital objects and continued accessibility of their content. The former requires the long-term maintenance of the bitstream of digital objects (i.e. backing up files), while long term access is decomposed into a set of goals that the process of digital preservation is intended to ensure:

a. *viability* is the reassurance that digital objects are intact and readable from the storage media. Since technology changes, media obsolescence is becoming a big problem. Assume we want to retrieve our documents from a 5.25" floppy disk. Nowadays it is difficult to find a computer with a 5.25" floppy drive.

b. *renderability* means that a digital object can be used in the way it was intended. Renderability is threatened by the obsolescence of software applications. For example two decades ago Wordstar files had the largest market penetration, but only a few people today can read any of the millions of Wordstar files that exist.

c. *intelligibility* is the reassurance that digital objects are properly interpreted by humans. So even if we are able to see the contents of a file, we cannot be sure that we will be able to understand it. So sufficient documentation should also be preserved to allow a user in future to understand the meaning of a digital object.

d. *fixity* is the quality of not being unintentionally altered or destroyed. Fixity can be threatened by insecure storage, transmission errors, or media degradation. It is particularly important for digital objects, because unlike analog objects, even a single destroyed bit can cause the entire object unusable.

e. *authenticity* refers to the trustworthiness of the digital objects. Authenticity is enhanced if it is possible to verify the creator of the digital object and establish that the object has not been changed. The documentation of the creation and the derivation history of a digital object is often known as digital provenance and if it is itself trustworthy, goes a long way towards verifying the authenticity of the object.

For example suppose that we want to preserve a collection of files containing observations of rainfall, temperature, pressure and wind velocities of various places on earth. More specifically every line from these files will contain the coordinates of the measurement location and the measurement values (in a specific order). Apart from preserving the bits of these files we must also preserve the ability to render them. To this end we must identify how the bit sequences are translated to text and therefore render this text on screen. However even if these data are properly rendered, the users will view some numbers without any indication about the proper interpretation of these numbers. So we should also preserve extra information that will aid users to understand these data. Furthermore it is very important to capture also information regarding the provenance of the digital objects of this collection. For example if these files have been derived from other "primitive" data, using specific computational processes, then the knowledge of these processes and the initial and intermediate data is necessary since it will allow the validity and reproducibility of the collection of files. Additionally provenance guides the answering of queries of form: Who created the data product and how? Which were the parameters of the processes that were used? Which were the initial data that were used?

Synopsizing, the objective of digital preservation is to ensure on the long run the maintainability and accessibility of a digital object.

1.2 Contribution of this thesis

In this thesis our focus concentrates on the preservation of the intelligibility and the preservation of provenance. The contribution of this thesis lies in:

- Proposing a formal model for representing and managing the intelligibility of digital objects comprising the abstract notions of module, dependency and DC profile.
- Specifying a number of basic intelligibility-related services
- Providing the guidelines for modeling and implementing the model for the preservation of intelligibility of digital objects.
- Describing the modeling and querying requirements for the preservation of provenance of digital objects
- Automating the creation and maintenance of preservation metadata.
- Reporting our experiences from the application of these models and tools in real-world data sets.

The results of this thesis have been published in [30, 40, 44, 43]

1.3 Organization of this thesis

Chapter 1 is the introductory chapter of the thesis.

Chapter 2 elaborates with models and languages for the preservation of digital objects.

Chapter 3 introduces a formal model for the preservation of intelligibility of digital objects based on dependency management.

Chapter 4 elaborates on the problem of modeling provenance.

Chapter 5 describes a tool that automates the ingestion and transformation of metadata of digital files.

Chapter 6 concludes and report possible ideas that are worth for further research.

Chapter 2

Digital Preservation Models and Languages

This chapter describes models and languages that could be employed for Digital Preservation. It begins with a description of the conceptual model of the Open Archival Information System (OAIS) in Section 2.1. Subsequently, at Section 2.2, several languages and formats (EAST, DEDSL, XFDU, Semantic Web Languages) are described and is discussed how they could be used for the preservation of digital objects.

2.1 OAIS Reference Model

An Open Archival Information System (for short OAIS) is an archive consisting of an organization of people and systems, that has accepted the responsibility to preserve information and make it available for a Designated Community. The term OAIS also refers to the ISO reference model [20] defined by a recommendation of the Consultative Committee for Space Data Systems[1]. This reference model, among others, provides a framework for the understanding and increased awareness of archival concepts needed for long term digital information preservation and access. The reference model address a full range of archival information preservation functions including ingest, archival storage, management, access and dissemination. The major purpose of OAIS reference model is to facilitate a much wider understanding of what is required to be preserved.

According to OAIS reference model *Information* is defined as any piece of knowledge that

[1] http://www.ccsds.org

is exchangeable and can be expressed by some type of data. For example the information in a Greek novel book is expressed as the characters that are combined to create text. The same pattern occurs in digital world. The information carried out in a text file in digital form (in a CD-ROM) is expressed by the bits it contains which, when they are combined with extra information that interprets them (i.e. mapping tables and rules) will convert them to more meaningful representations (i.e. characters). This extra information is called Representation Information. Every Data Object accompanied with Representation Information that interprets it, form an Information Object. A *Data Object* is either a physical object (an object with physically observable properties) or a digital object (a sequence of bits). *Representation Information (RI)* is the information that maps a Data Object into more meaningful concepts. 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. It follows that intelligibility is closely related to RI. In the previous example the extra RI we should store is the information about the proper mapping of bit sequences to characters.

Representation Information however is just another Information Object, which means that it is typically composed of its own data and other RI. This recursive nature of RI typically leads to a potentially long chain of RI objects, called Representation Network. However OAIS aims at preserving information for a Designated Community and therefore must take into account the knowledge that is assumed to be known by the community. In terms of OAIS it is called Knowledge Base. A person or system can be said to have a *Knowledge Base* which allows them to understand received information. For example assume that the Greek novel of the previous example exists in digital form, it is a text file. The information stored within the file is expressed by the bits (the actual data). However these data are not useful without the existence of appropriate Representation Information. The combination of the Data Object (the file) with its Representation Information (mapping of byte codes to characters) make this file understandable to the recipient's Knowledge Base (the knowledge of Greek language and grammar).

Information is submitted to an OAIS, or disseminated to a consumer through Information Packages. An *Information Package* consists of the Content Information and Preservation Description Information. The Content Information consists of the Data Object and its associated Representation Information. Preservation Description Information (PDI) applies to the Content Information and is necessary for adequate preservation of the content information. PDI is

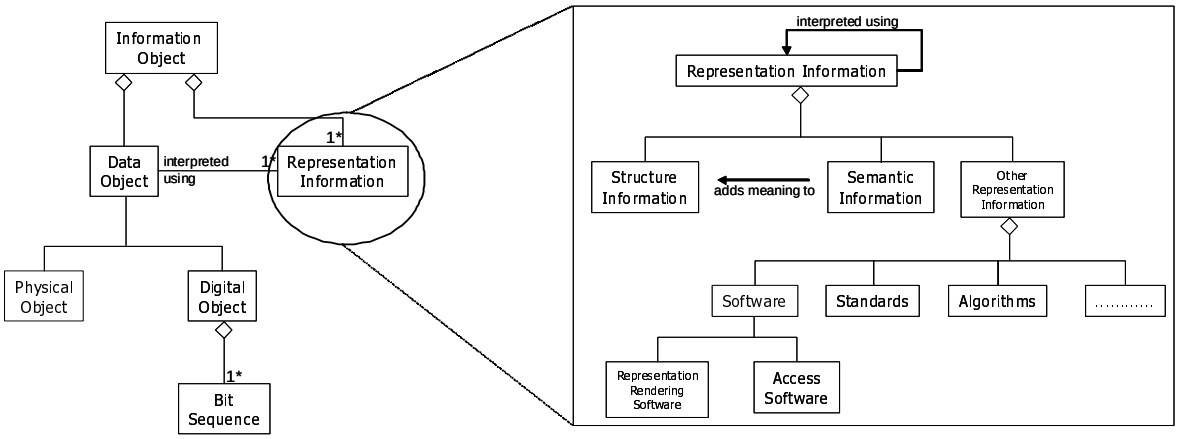

Figure 2.1: The Information Model of OAIS

further divided in four types of preservation information, Provenance, Context, Fixity. Additionally it is necessary to distinguish packages that are used for preservation by an OAIS and packages that are used for submission and dissemination. This policy reflect the reality that some submissions to OAIS might have insufficient PDI to meet final OAIS requirements or we might want to deliver different (with more or less PDI) Information Packages to different communities (having different Knowledge Bases). Therefore Information Packages are categorized to *Submission Information Packages (SIP)* which are packages sent to an OAIS by a producer, *Archival Information Packages (AIP)* which is a package containing a full set of PDI and is intended for preservation within an OAIS, *Dissemination Information Packages (DIP)* that are information packages delivered to consumers.

It's important to notice that OAIS accommodates information that is inherently non-digital but the modeling and preservation of such information is not addressed in detail. We will use these notions in the sequel (in the formalization of the intelligibility model).

2.2 Languages for Digital Preservation

We will describe how the various languages and formats (EAST, DEDSL, XFDU, Semantic Web languages) could be used for the preservation of digital files. We will use a collection of files containing temperature measurements from various places on earth as a running example and we will discuss the pros and cons of each approach.

Suppose we want to preserve the files containing temperature measurements from various

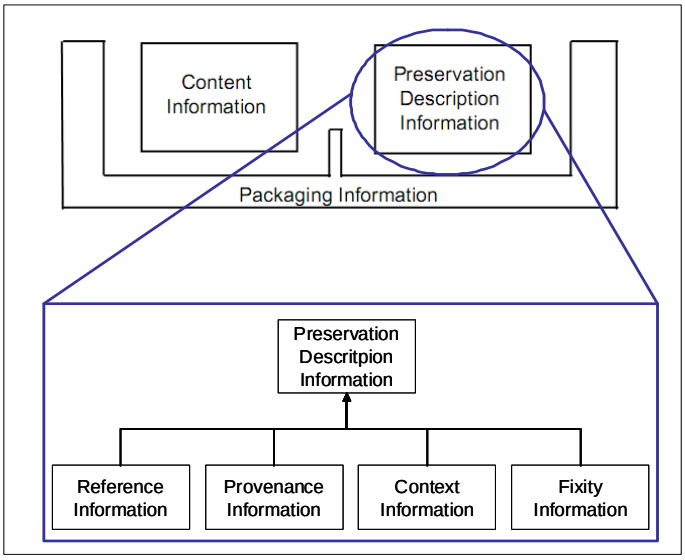

Figure 2.2: Packaging Information and Preservation Description Information

places on earth. Each file comprises an arbitrary number of lines where each line contains three numerical values corresponding to the longitude, the latitude and the measured temperature (in Celsius degrees). Each line corresponds to the temperature at the coordinate-specified area as it was measured at a certain point in time. The time of measurement is hardwired in the name of the file, e.g. a file named `datafile20080903_12PM.txt` is supposed to contain measurements taken at 12pm of the 3^{rd} September of 2008. Suppose that the contents of this file are:

```
25.130    35.325    30.2
25.100    35.161    28.9
25.180    35.333    29.3
```

We may have several such files (all having the same format though) each one containing measurements at different locations and times.

2.2.1 Preserving Structure

In order to preserve the structure of the `datafile20080903_12PM.txt` we could make use of the EAST (Enhanced Ada Subse T)[26] language. Each data description record (DDR), according to that language, consists of two packages, one for the logical description and one for physical description of the data. These two packages are mandatory even if the content of the physical part is empty. The first package includes a logical description of all the described

components, their size in bits as well as their location within the set of the described data. The physical part includes a representation of some basic types defined in the logical description and are dependent on the machine that generates these data: the organization of arrays (i.e. first-index-first, last-index-first) and the bit organization on the medium (high-order-first or low-order-first for big-endian or little endian representation accordingly). Figure 2.3 shows an example of a DDR describing `datafile.txt`. We defined three different types one for each column of a data file (longitude, latitude , temperature), since each column represents a different kind of data. More precisely the distinction of longitude and latitude is only made because of their different upper and lower limits.

```
package logical_datafileX_description is

type HORIZONTAL_COORDINATE is range -90.00 .. 90.00
for HORIZONTAL_COORDINATE'size 64; --bits

type VERTICAL_COORDINATE is range -180.00 .. 180.00
for VERTICAL_COORDINATE'size 64;

type TEMPERATURE_TYPE is range -100.0 .. 200.0
for TEMPERATURE_TYPE'size 16;

type MEASUREMENT_TUPLE is record
    LONGITUDE:VERTICAL_COORDINATE
    LATITUDE:HORIZONTAL_COORDINATE
    MEASURED_TEMPERATURE:TEMPERATURE_TYPE
end record;
for MEASUREMENT_TUPLE'size use 144;

type MEASUREMENT_BLOCK is array(1..1000) of MEASUREMENT_TUPLE;
for MEASUREMENT_BLOCK'size use 144000;

SOURCE_DATA:MEASUREMENT_BLOCK

end logical_datafileX_description;

package physical_datafileX_description is

end physical_datafileX_description;
```

Figure 2.3: An example of an EAST description

2.2.2 Preserving Semantics

We could define semantic descriptions for the entities (longitude, latitude and temperature) of the file `datafileX.txt` aiming at preserving the meaning (clarifying the interpretation) of

the terms "longitude" "latitude" and "temperature".

We could use the DEDSL (Data Entity Dictionary Specification Language)[25] language. Figure 2.4 shows an example of a DEDSL description for `datafileX.txt` according to the implementation of DEDSL using XML.

Note that if we have another file with the same kind of information, we could reuse the same semantic descriptions, so semantic descriptions can be considered as reusable models. These models can contain abstract data descriptions to which concrete descriptions may refer.

2.2.3 Packaging

Now suppose that we want to preserve and archive a number of datafiles with such measurements. Packaging formats could be used for preparing a package that contains the data files plus their EAST and DEDSL descriptions. We could satisfy such packaging requirements using **XFDU** (XML Formated Data Unit (XFDU) [11, 28] which is a standard file format developed by CCSDS (Consultative Committee for Space Data Systems) for packaging and conveying scientific data, aiming at facilitating information transfer and archiving. The benefits of adopting a packaging approach (like that of XFDU) is that we can also add various information about the components of the package. For example suppose we would like to include information about the user that took the temperatures for each file, as well as the GPS (Global Positioning System) and the thermometer characteristics or the satellite information (if the samplings were made from space). We can easily add the above information using XFDU since we just have to add the necessary information at the package. The major benefit from the use of XFDU is that we can package together heterogenous modules (java programs, datafiles, GPS info, provenance data) and deliver them to the user, or archive them, as a single (ideally self-describing) unit.

As another example, suppose we have a software application, e.g. a java program, that calculates the daily, weekly and monthly average temperatures of various locations. To make such aggregate calculations the program needs a number of files containing the data (datafiles). For example in order to calculate the daily average temperatures for each location for the day 3^{rd} September of 2008 we need all data files of the form datafile20080903_$****$.txt. For delivering this program to a user, as an application that contains past measurements, we have to provide her with the required data files. In this case, the XFDU package should contain both the java.code and the files containing the data. Figure 2.5 shows an example of such a package (for reasons of brevity we depict only two data files), while Figure 2.6 illustrates the structure

```
<?xml version="1.0" encoding=="UTF-8"?>

<DATA_ENTITY_DICTIONARY>
    <DICTIONARY_IDENTIFICATION>
        <DICTIONARY_NAME CASE_SENSITIVITY="NOT_CASE_SENSITIVE">datafileX Dictionary
        </DICTIONARY_NAME>
    </DICTIONARY_IDENTIFICATION>

    <DATA_ENTITY_DEFINITION CLASS="DATA_FIELD" NAME="LONGITUDE">
        <DEFINITIONAL_PART>
            <DEFINITION>
                It represents the longitude for some certain coordinates. Longitudes east of
                Greenwich shall be designated by the use of plus (+) symbol while longitudes
                west of Greenwich shall be designated with the use of minus (-) symbol.
            </DEFINITION>
            <SHORT_DEFINITION>Longitude</SHORT_DEFINITION>
            <UNITS>deg</UNITS>
        </DEFINITIONAL_PART>
        <REPRESENTATIONAL_PART DATA_TYPE="REAL">
            <RANGE MIN="-180.00" MAX="+180.00"/>
        </REPRESENTATIONAL_PART>
    </DATA_ENTITY_DEFINITION>
    <DATA_ENTITY_DEFINITION CLASS="DATA_FIELD" NAME="LATITUDE">
        <DEFINITIONAL_PART>
            <DEFINIITION>
                It represents the latitude for some certain coordinates.  Latitudes north of the Equator
                shall be designated by the use of plus (+) symbol while latitudes south of the Equator
                will be designated by the use of minus (-) symbol.
            </DEFINITION>
            <SHORT_DEFINITION>Latitude</SHORT_DEFINITION>
            <UNITS>deg</UNITS>
        </DEFINITIONAL_PART>
        <REPRESENTATIONAL_PART DATA_TYPE="REAL">
            <RANGE MIN="-90.00" MAX="+90.00"/>
        </REPRESENTATIONAL_PART>
    </DATA_ENTITY_DEFINITION>
    <DATA_ENTITY_DEFINITION CLASS="DATA_FIELD" NAME="TEMPERATURE">
        <DEFINITIONAL_PART>
            <DEFINITION>
                It represent the temperature in the area with
                the specific coordinates.
            </DEFINITION>
            <SHORT_DEFINITION>Temperature</SHORT_DEFINITION>
            <UNITS>Celsius degrees</UNITS>
        </DEFINITIONAL_PART>
        <REPRESENTATIONAL_PART DATA_TYPE="REAL">
            <RANGE MIN="-100.0" MAX="200.0"/>
        </REPRESENTATIONAL_PART>
    </DATA_ENTITY_DEFINITION>
</DATA_ENTITY_DICTIONARY>
```

Figure 2.4: An example of a DEDSL description

of such a package.

```
<?xml vesrion="1.0" encoding="UTF-8"?>

<xfdu:XFDU xmlns:xfdu="http://www.ccsds.org">
  <packageHeader ID="id"/>
  <informationPackageMap ID="id">
    <xfdu:contentUnit ID="file:/avgCalculator.java">
    <dataObjectPointer dataObjectId="dataObjectAC"/>
      <xfdu:contentUnit ID="file:/data">
        <dataObjectPointer dataObjectID="dataObject20080903_12PM"/>
        <dataObjectPointer dataObjectID="dataObject20080903_12AM"/>
      </xfdu:contentUnit>
     </xfdu:contentUnit>
  </informationPackageMap>
  <dataObjectSection>
    <dataObject size="5620" ID="dataObjectAC">
       <byteStream size="5620" mimeType="text/plain">
          <fileLocation href="file:/avgCalculator.java"/>
       </byteStream>
    </dataObject>
      <dataObject size="122108" ID="dataObject20080903_12PM">
        <byteStream size="122108" mimeType="text/plain">
          <fileLocation href="file:/data/datafile20080903_12PM.txt"/>
        </byteStream>
      </dataObject>
      <dataObject size="134554" ID="dataObject20080903_12AM">
        <byteStream size="134554" mimeType="text/plain">
          <fileLocation href="file:/data/datafile20080903_12AM.txt"/>
        </byteStream>
      </dataObject>
  </dataObjectSection>
</xfdu:XFDU>
```

Figure 2.5: An example of a XFDU package

2.2.4 Modeling using Semantic Web Languges

Semantic Web (SW) languages could be used in order to preserve these data files. The adoption of SW languages (like RDF/S) have an additional benefit. A top/upper level ontology could be used to describe (syntactically and semantically) the form of the data and other files could instantiate this ontology. These instantiations are actually the data themselves. When creating a new data file there is no need to create its DEDSL or EAST description every time, but in contrary to write the data with the syntax specified by the ontology. Moreover whenever more data types are needed to be preserved the ontology can be easily extended. For example, past data files can contain only the longitude, the latitude and the temperature, while current ones may contain also the name of each location, or the thermometer used for the measurement.

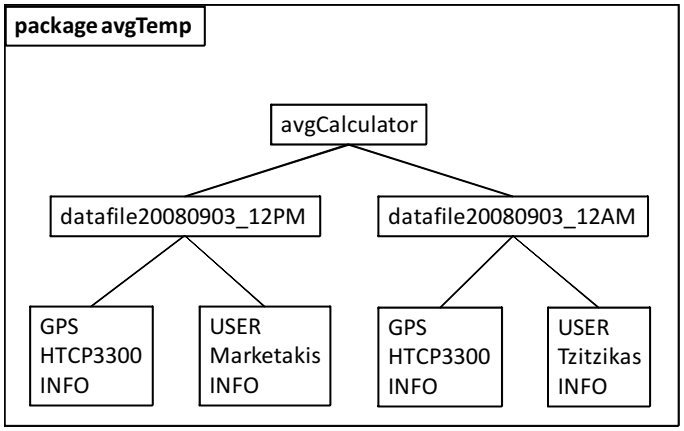

Figure 2.6: Structure Organization for a XFDU package

In such cases two different kinds of DEDSL/EAST descriptions have to be created and used. In the SW approach, we just have to extend the ontology. Figure 2.7 shows an indicative ontology for our running example expressed in RDF/S XML. Other ontologies of wider scope could be used as well (e.g. CIDOC CRM [21]).

```
<?xml version='1.0'?> <rdf:RDF xml:lang="en"
         xmlns:rdf="http://www.w3c.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3c.org/2000/01/rdf-schema#">

    <rdfs:Class rdf:ID="Sample"/>
    <rdf:Property rdf:ID="Longitude">
        <rdfs:domain rdf:resource="#Sample"/>
        <rdfs:range rdf:resources="&rdfs;Literal"/>
    </rdf/Property>
    <rdf:Property rdf:ID="Latitude">
        <rdfs:domain rdf:resource="#Sample"/>
        <rdfs:range rdf:resource="&rdfs;Literal"/>
    </rdf:Property>
    <rdf:Property rdf:ID="Temperature">
        <rdfs:domain rdf:resource="#Sample"/>
        <rdfs:range rdf:resource="&rdfs;Literal"/>
    </rdfs:Property>
</rdf:RDF>
```

Figure 2.7: Example of an ontology for measurements expressed in RDF/S

Another benefit of the SW languages is that the data and their descriptions are tightly coupled. When using EAST/DEDSL the descriptions are extracted from the form of the actual data while the file containing the data and the file describing them are two distinct files. Due to this shortcoming, packaging formats are important. On the other hand in RDF a data file would itself define the data type of each data element in the file. To make this more obvious let's take

a line from our running example. The line `25.130 35.325 30.2` does not offer the knowledge of what `25.130` might be, it could be either the longitude, the latitude or the temperature. On the other hand this information using RDF could be

```
<temperature:Sample
      rdf:about="sampingId20080903_12PM_35.233_25.343>
    <temperature:Longitude= "25.130"/>
    <temperature:Latitude= "35.325"/>
    <temperature:Temperature= "30.2"/>
</temperature:Sample>
```

This example shows clearly that one can understand easily what is the purpose of each variable without the existence of another file that describes the data (like in EAST/DEDSL).

Chapter 3

Preservation of Intelligibility of Digital Objects

Digital preservation can be described as the set of actions that are necessary to ensure the long-term maintenance of digital objects and the continued accessibility of their content. Among others the long-term accessibility includes the task of ensuring that digital objects remain intelligible. Intelligibility refers to the ability of humans to understand what a digital object is, how they can use it, what is the conceptualization of the object. Therefore the preservation of intelligibility might require additional information, that sufficiently document a digital object and allows one to understand and interpret it properly.

In this chapter we identify the several aspects of digital objects as well as what kind of information (and how much of it) do we need in order to ensure that a digital object remains intelligible. We reduce the problem of deciding intelligibility to dependency management between digital objects. We discuss some OAIS related concepts and result in the proposal of a formal model for the preservation of intelligibility of digital objects on the basis of designated community knowledge and we identify a set of basic intelligibility-related services based on this model. Furthermore we provide the guidelines for the modeling and implementation of the model. Finally we describe the design and implementation of the `GapMgr` tool that realizes these notions, and report our experiences from the application of this model in the context of CASPAR project.

3.1 On Digital Objects and Dependencies

We live in a digital world. Everyone nowadays work and communicate using computers. We communicate digitally using e-mails and voice platforms, watch photographs in digital form, use computers for complex computations and experiments. Moreover information that previously existed in analog form (i.e. paper) is digitized. The amount of digital objects that libraries and archives maintain constantly increases. It is therefore urgent to ensure that these digital objects will remain functional, usable and intelligible in future. But what should we preserve and how? To answer this question we must first define what a digital object is.

A Digital object is an object composed of a set of bit sequences. At the bit stream layer there is no qualitative difference between digital objects. However in upper layers we identify several types of digital objects. They can be classified to simple or composite, static or dynamic, rendered or non-rendered etc. Since we are interested in preserving the intelligibility of digital objects we can distinguish them in two broad categories based on the different interpretations of their content, to information objects and computational objects. Information objects contain the knowledge about something in a data structure that allows their exchangeability. Examples of information objects are documents, data-products, images, ontologies. The content of information objects can many times be described straightforwardly in analog form, i.e. the contents of a document are the same in reader's eyes in digital form and printed on a paper. Computational objects on the other hand are actually sets of instructions for a computer. These objects use computational resources to do various tasks. Typical examples of computational objects are software applications.

Information objects are absolutely useless if we cannot understand their content. So information objects may require extra information that allows one to understand them. This extra information is expressed using other information objects. For example in order to understand the concepts of an ontology we must also have available any other concept from any other ontology it uses. As another example consider scientific data products. These products are usually aggregations of other (even heterogeneous) primitive data. The provenance information of these data products can be preserved using scientific workflows [9]. The key advantage of scientific workflows is that they record data (and process) dependencies during workflow runs. For example Figure 3.1 shows a workflow that generates two data products. In this figure rectangles represent data and the edges are used to capture derivative data and express the dependability of the final products from the initial and intermediate results. Capturing these dependencies

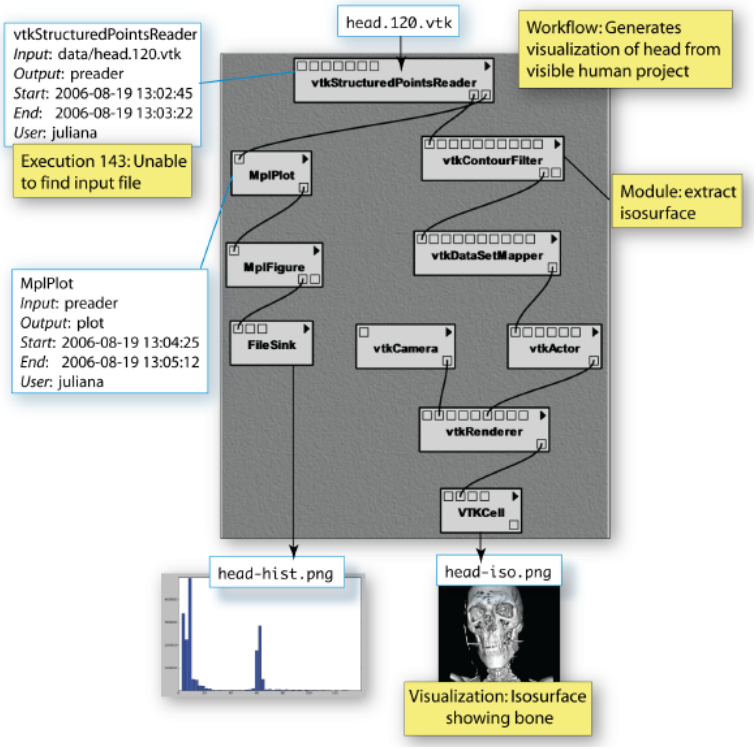

Figure 3.1: The generation of two data products as a workflow

allow the understandability, reproducibility and validity of data products.

Information objects are typically organized as files (or collections of files). It is not possible however to identify the contents of a file if we do not have the software that was used to create it. Indeed information objects use complex structures to encode information and embed extra information that is meaningful only to the software application that created them. For example a MS-Word document embeds special formatting information about the layout, the structure, the fonts etc. This information is not identifiable from another text editor, i.e. Notepad. As a result information objects have become dependent from software.

Software applications in turn typically use computational resources to perform a task, for example a java program that does complex mathematic calculations. The above program does not explicitly use these resources (i.e. it does not define memory addresses to store numbers), but rather exploit the functionalities of other programs that handles these issues. Furthermore software reusability allows one to reuse a software program and exploit its functionalities. Although software re-use is becoming a common practice, this policy results to dependencies

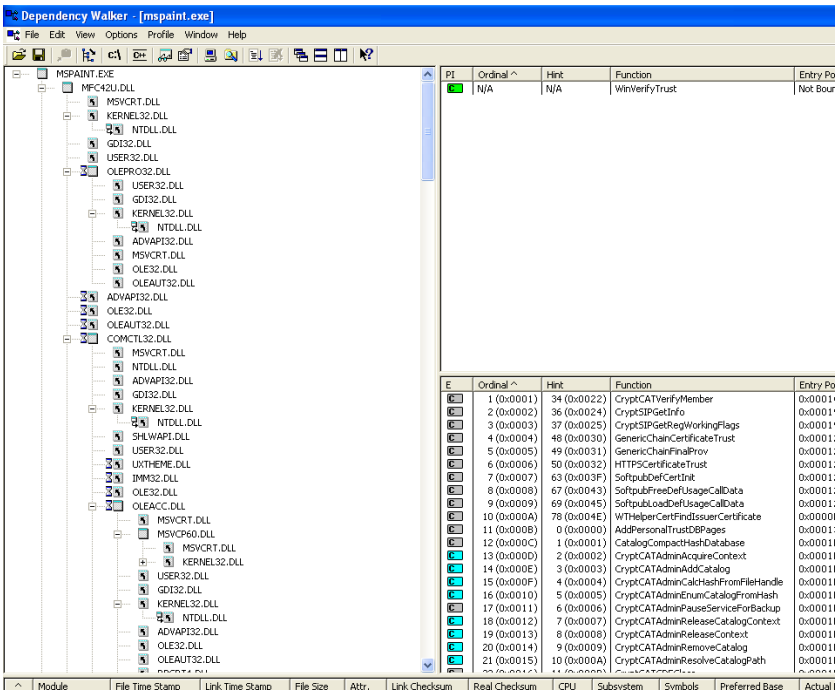

Figure 3.2: The dependencies of mspaint software application

between software components. These dependencies are interpreted as the reliance of a software component on others to support a specific functionality. In other words a software application cannot even function if the applications it depends on are not available, i.e. we cannot run the above java program if we haven't installed a Java Virtual Machine.

It is becoming clear that digital objects are actually complex data structures that either contain information about something or use computational resources to do various tasks. These objects depend on a plethora of other resources whose record is of great importance for the preservation of their intelligibility. Additionally these dependencies can have several different interpretations. In [38, 14] dependencies between information objects are exploited for ensuring consistency and validity. In [9] dependencies are used over data products, through scientific workflows. Such dependencies apart from being useful for understanding and validating data can also aid the reproducibility of data products. [2] proposes an extended framework for file systems that allow users define dependency links between files. The semantics of these links are defined by users through key-value pairs. Finally many dependency management approaches for software components [5, 15, 46, 47, 48] have been described in the literature. The interpretation of these dependencies vary including the safety of installation or de-installation, the ability to

perform a task, the selection of the most appropriate component, the consequences in software components after a specific component's service is called.

3.1.1 OAIS - Preserving the Understandability

According to OAIS *Data Objects* are considered to be either physical objects (objects with physically observable properties) or digital objects. Every Data Object along with some extra information about the object forms an Information Object. *Information* is defined as any piece of knowledge that is exchangeable and can be expressed by some type of data. For example the information in a Greek novel book is expressed as the characters that are combined to create text. The information carried out in a text file in digital form (in a CD-ROM) is expressed by the bits it contains which, when they are combined with extra information that interprets them (i.e. mapping tables and rules) will convert them to more meaningful representations (i.e. characters). This extra information that maps a Data Object into more meaningful concepts is called *Representation Information (RI).* 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. It follows that intelligibility is closely related to RI.

Every Information Object needs a RI object that interprets it. This relationship (*interpreted using*) is actually a specialized form of dependency between Information Objects. The semantics of this dependency is to explain in a human-understandable manner how a data object can be interpreted. For example if a data object is a digital image then its RI will describe how the content of the image can be rendered on a computer screen. Since RI is intended for humans the above information must be in a human-understandable form. It does not address the issue of explicitly denoting that an information object (i.e. an image) needs a specific application that recognizes it (i.e. an image viewer). It's a responsibility of an OAIS to find (or create new) software conforming to the given Representation Information for this information object.

The notion of interpretation as stated by OAIS is more restricted than the general notion of dependency, described in the previous section. Dependencies can be exploited for capturing the required information for digital objects not only in terms of their intelligibility, but also in terms of validity, reproducibility or functionality (if it is a software application). Furthermore the interpretation of digital objects using human-understandable information may not be always feasible.

For example a file in FITS format is understandable only from persons that know how to

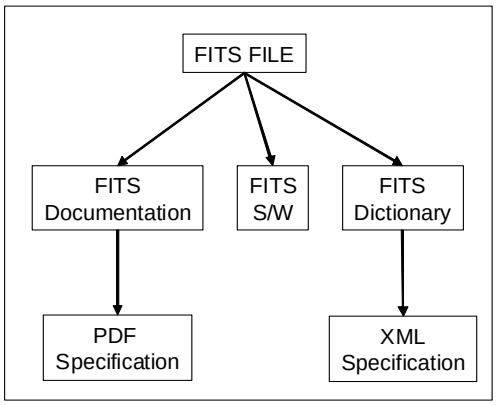

Figure 3.3: The Representation Network of a FITS file

handle this format. Someone that does not understand this file needs additional Representation Information. Figure 3.3 shows such a representation network for FITS file. Rectangles are used to denote RI objects and edges are used to denote dependencies of the form *interpretedUsing*. Consequently the extra information that is required by someone that does not know anything about FITS files would be information about the software that is needed and the concepts and notations that are used from FITS files. More specifically the RI for FITS software describes which are the algorithms that are used and recognize FITS files. So it is actually a description of the application in human understandable form. Additionally the FITS Documentation and FITS Dictionary are expressed in PDF and XML respectively. So we must provide extra RI that interprets them.

3.2 A Formal Model for the Intelligibility of Digital Objects

3.2.1 A Core Model for Digital Objects and Dependencies

We introduce a core model for the intelligibility of digital objects based on dependencies. The basic notions of the model are the notions of *Module* and *Dependency*. As we described in Section 3.1 digital objects are dependent from a plethora of other resources. Recall that these resources can be described as objects containing information and objects using other resources. We use the very general notion of module to model these resources. A **module** can be any digital object, either information or computational object. There is no standard way to define a module, so we can have modules of various levels of abstraction, a module can be a collection

of documents, or alternatively every document in the collection may be a module.

In order to ensure the intelligibility of digital objects we must first identify which are the permissible actions with these objects. This is very important due to the heterogeneity of digital objects and the information they convey. For example assuming a file `HelloWorld.java`, some indicative actions we can perform with it is to compile it and read it. To this end we introduce the notion of tasks; A **task** is any action that can be performed with modules. Once we have identified the tasks, it is important, for the preservation of intelligibility, to preserve how these tasks are performed. So we have to define which are the necessary resources (in terms of modules) for the performability of a task using the notion of dependencies. The reliance of a module on others for the performability of a task is denoted using **dependencies** between modules. For example for the task of compiling the previous file we could define that it depends on the availability of a java compiler and any other classes, libraries it requires (i.e. import statements). Therefore a rich dependency graph is created over the set of modules. In the sequel we use the following notations; we define T as the set of all modules and the binary relation $>$ on T is used for representing dependencies. A relationship $t > t'$ $(t, t' \in T)$ means that t depends on t'.

The interpretations of modules and dependencies are very general and can capture a plethora of cases. Additionally the determination of dependencies based on tasks will lead to the creation of several dependencies, with different interpretations. In order to distinguish the various interpretations of dependencies we enrich the notion of dependencies with dependency types. We require every dependency to be assigned with at least one type that denotes which is the objective of the dependency. For example the software dependencies in Figure 3.2 (Section 3.1) denote dependencies of a software application if we want to run it. Therefore these dependencies should be assigned an appropriate type (i.e. `_run`). Complementarily we can organize dependency types hierarchically. The motivation of such a taxonomy of types is the need for enabling deductions of the form "if we can do task A then certainly we can do task B". For example if we can edit a file then certainly we can read it. Therefore let D be the set of all dependency types, two types $d, d' \in D$ then $d \sqsubseteq d'$ if d is a subtype of d', i.e. `_edit` $\sqsubseteq$ `_read`.

Module types are defined in a similar manner. These types can be exploited to specialize the very general notion of module to more concrete elements. For example a module containing the source code of a program in java could be defined as `SoftwareSourceCode`. However every module that contains source code can be defined also as a module containing text. Therefore

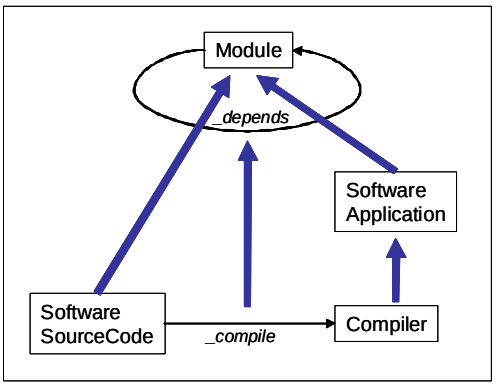

Figure 3.4: Restricting the domain and range of dependencies

the above type is actually a subtype of `TextFile`. So again we can organize module types as an hierarchy of types. If C is the set of all modules and $c, c' \in C$ then $c \sqsubseteq c'$ denotes the subtype relation in C.

Since the number of dependency types may increase, it is beneficial to organize them following an object-oriented approach. More specifically we can use module types and specify the domain and range of each dependency type. For example consider the dependency type `_compile` denoting the dependencies regarding the compilation of a file. Clearly the domain of this dependency type must be a module denoting source code files while the range must be a compiler. This scenario is shown in Figure 3.4. Thick arrows are used to denote subtype relationships.

Our model allows the representation of dependencies of various interpretations. Just indicatively we can model all the dependencies of digital objects as presented at Section 3.1. For example in Figure 3.1 a data workflow example is illustrated. This workflow defines the derivation of data product from its initial data to allow the understandability, validity and reproducibility of the final data product. In our model data can be modeled as modules and dependencies can be exploited to model the derivation edges, in order to preserve the above tasks. Similarly our model allows a straightforward modeling of a representation network of OAIS (Figure 3.3), since the notion *interpretedUsing* is just a specialized form of dependency. Additionally a module can have several dependencies of various types. For example Figure 3.5 illustrates the dependencies of a file in FITS format. Modules are represented as boxes and arrows between them are used to denote dependencies (the starting module depend on the ending module) of different types. The file `mars.fits` can be read only with an appropriate application which is modeled with

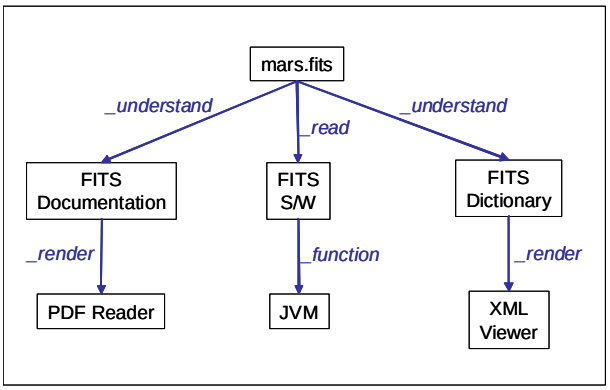

Figure 3.5: Modeling the dependencies of a FITS file

the `FITS S/W` module. This in turn requires the availability of Java Virtual Machine in order to function. If our aim is to understand the concepts of the file the we must have available the FITS Documentation and FITS Dictionary. These files are in PDF and XML format and therefore require the existence of the appropriate application that renders their contents. Such a dependency graph is richer than an OAIS-representation network (recall Figure 3.3), since it allows the definition of dependencies of various interpretations.

3.2.1.1 Conjunctive versus Disjunctive Dependencies

The dependencies (and their types) are determined by the task that must be performed with modules. More specifically for a given module we identify which are the dependencies of the module in order to perform a specific task with it. However there are usually more than one ways to perform a task. Consider for example the file `HelloWorld.java`. For ensuring its readability, a text editor is required, i.e. `NotePad`. However this file can also be read using other text editors (i.e. `VI`). Therefore the dependencies regarding the readability of the file should be that `HelloWorld.java` depends on `NotePad` **OR** `VI`. Dependencies are considered so far to have conjunctive dependencies, so it is not possible to model the above scenario. We must address the problem of determining dependencies also in a disjunctive manner.

The disjunctive nature of dependencies was first approached at [41] using the concept of generalized modules, a set of modules that were interpreted disjunctively. The management of these modules was rather complex, so in the context of this work we found an extended model by allowing disjunctive dependencies using Horn rules. The extended model apart from simplifying the disjunctive semantics of dependencies is also more expressible and flexible, as it

allows defining properties of dependencies straightforwardly.

Suppose that we want to preserve the digital objects of a user's laptop. The user's system contains the files `HelloWorld.java`, `HelloWorld.cc` and the software components `javac`, `NotePad`, `VI` and `JVM`. We want to preserve the ability to edit, read, compile and run these files. To this end we are using facts and rules. The digital objects of the system are modeled using facts while rules are employed in order to represent tasks and dependencies. Moreover we can use rules to represent module and dependency type hierarchies (i.e. `JavaSourceCode` $\sqsubseteq$ `TextFile`, `_edit` $\sqsubseteq$ `_read`). Below we provide the set of facts and rules that hold for the above example in a human readable form.

1. `HelloWorld.java` is a `JavaSourceFile`
2. `HelloWorld.cc` is a `C++SourceFile`
3. `NotePad` is a `TextEditor`
4. `VI` is a `TextEditor`
5. `javac` is a `JavaCompiler`
6. `JVM` is a `JavaVirtualMachine`
7. Every `JavaSourceFile` is also a `TextFile`
8. Every `C++SourceFile` is also a `TextFile`
9. A `TextFile` is `Editable` if there is a `TextEditor`
10. A `JavaSourceFile` is `JavaCompilable` if there is a `JavaCompiler`
11. A `C++SourceFile` is `C++Compilable` if there is a `C++Compiler`
12. A file is `Readable` if it is `Editable`
13. A file is `Compilable` if it is `JavaCompilable`
14. A file is `Compilable` if it is `C++Compilable`

Lines 1-6 are actually facts describing the digital objects while lines 7-14 are rules denoting various tasks and how they can be carried out. More precisely the rules 7,8 are used to denote an hierarchy of module types (`JavaSourceFile` $\sqsubseteq$ `TextFile` and `C++SourceFile` $\sqsubseteq$

`TextFile`) and the rules 12-14 are used to define an hierarchy of tasks (`Editable` $\sqsubseteq$ `Readable`, `JavaCompilable` $\sqsubseteq$ `Compilable` and `C++Compilable` $\sqsubseteq$ `Compilable`). Finally the rules 9-11 are used to express which are the tasks that can be performed and which are the dependencies of those tasks (i.e. the readability of a `TextFile` depends on the availability of a `TextEditor`). Using such facts and rules we model the modules and their dependencies based on the tasks that can be performed. For example in order to determine the compilability of `HelloWorld.java` we must use the rules 1,5,10,13. In order to read the content of the same files we must use the rules 1,3,7,9,12. Alternatively (since there are 2 text editors) we can perform the same task using the rules 1,4,7,9,12.

Below we provide the formal definition of modules, dependencies, the semantics of these dependencies, as well as various properties of these concepts. We will use the terminology and syntax used in Datalog to address these issues.

Modules - Module Type Hierarchies Modules are expressed as facts. Since we allow a very general interpretation of what a module may be there is no distinction between information objects and software components. We define all these objects as modules of an appropriate type. For example the digital objects of the previous example are defined as follows:

```
JavaSourceFile('HelloWorld.java').
C++SourceFile('HelloWorld.cc').
TextEditor('NotePad').
TextEditor('VI').
JavaCompiler('javac').
JavaVirtualMachine('JVM').
```

A Module can be classified to one or more modules types. Additionally these types can be organized hierarchically. Such taxonomies can be represented with appropriate rules. For example the source files for Java and C++ are also TextFiles, so we use the following rules:

```
TextFile(X) :- JavaSourceFile(X).
TextFile(X) :- C++SourceFile(X).
```

We can also capture several features of digital objects using predicates (not necessarily unary), i.e. `ReadOnly('HelloWorld.java').`

```
LastModifDate('HelloWorld.java', '2009-10-18').
```

Task - Dependencies - Dependency Type Hierarchies Tasks and their dependencies are modeled using rules. For every task we use two predicates; one (which is usually unary) to denote the task and another one (of arity equal or greater than 2) for denoting its dependencies. Consider the following example:

```
IsEditable(X) :- Editable(X,Y).
Editable(X,Y) :- TextFile(X), TextEditor(Y).
```

The first rule denotes that an object X is editable if there is any Y such that X is editable by Y. The second rule defines a dependency between two modules for this task. More precisely it defines that every TextFile depends on a TextEditor in order to edit its contents. Notice that if there are more than one text editors available (as here) then the above dependency is interpreted disjunctively (i.e. every TextFile depends on any of the two TextEditors). Relations of higher arity can be employed according to the requirements e.g.

```
IsRunnable(X) :- Runnable(X,Y,Z).
Runnable(X,Y,Z) :- JavaSourceFile(X), Compilable(X,Y), JavaVirtualMachine(Z).
```

Furthermore we can express hierarchies of types using rules. The motivation is the need for enabling deductions of the form "if we can do task A then we can do task B". For example if we can edit a file then certainly we can read it. This is expressed with the following rule: `Read(X) :- Edit(X).` Alternatively, or complementarily we can express such deductions at the dependency level: `Readable(X,Y) :- Editable(X,Y).` Finally we can express the properties of dependencies (e.g. transitivity) using rules. For example if the following two facts hold:

```
Runnable('HelloWorld.class', 'JavaVirtualMachine').
Runnable('JavaVirtualMachine', 'Windows').
```

then we might want to infer that the `HelloWorld.class` is also runnable under `Windows`. In order to define the dependencies of this task as transitive we must add the following rule: `Runnable(X,Z) :- Runnable(X,Y) , Runnable(Y,Z).` Other properties (i.e. symmetry) that dependencies may have are defined analogously.

3.2.1.2 Synopsis

Synopsizing we provide the formal definitions for modeling digital objects and their dependencies for the preservation of intelligibility.

- A Module can be any digital object (i.e. a document, a software application etc.) and may have one or more module types.
- The reliance of a module on others for the performability of a task is modeled using dependencies between modules.

Therefore the performability of a task determines which are the dependencies. In some cases a module may require all the modules it depends on while in other cases only some of these modules may be sufficient. In the first case dependencies have conjunctive semantics and the task can be performed only if all the dependencies are available while in the second case they have disjunctive semantics and the task can be performed in more than one ways (a text file can be read using `NotePad` **OR** `VI`).

Modules and dependencies form a dependency graph. The graph is global in the sense that it contains all the modules that have been recorded. Dependencies are represented as edges that connect the nodes-modules of the graph. Since the dependencies of a module are always the same the adoption of a global dependency graph is adequate. For example if we want to retrieve the dependencies of a module for the performability of a task we must find the module in the dependency graph and resolve its dependencies.

Axiom 1 *Modules and their dependencies form a dependency graph* $\mathcal{G} = (T, >)$. *Every module in the graph has a unique identifier and its dependencies are always the same.*

3.2.2 Formalizing Designated Community Knowledge

So far dependencies have been recognized as the key notion for preserving the intelligibility of digital objects. However since nothing is self-explaining we will result in long chain of dependent modules. So an important rising question is: how many dependencies do we need to record? OAIS address this issue by exploiting the knowledge that is assumed to be known from a community. According to OAIS a person, a system or a community of users can be said to have a *Knowledge Base* which allows them to understand received information (recall that OAIS aims at the human understandability of data objects). For example a person who has a Knowledge Base that includes the understanding of Greek language will be able to understand a Greek text.

We can use a similar approach and therefore limit the long chain of dependencies that have to be recorded, on the basis of the knowledge that is assumed to be known from a designated

community. A Designated Community is an identified group of users (or systems) that are able to understand a particular set of information. DC Knowledge is the information that is assumed to be known from the users of that community. The Knowledge Base of OAIS only makes assumptions about the community knowledge, i.e. Java programmers know how to program in JAVA. However in our work we allow making these assumptions *explicit* by assigning to the users of a community the modules that are known from them. To this end we introduce the notion of DC Profiles.

Definition 1 *A DC Profile $T(u)$ is a set of modules that are assumed to be known from the users u of a Designated Community.*

The above notion implies that the users of a community know a set of modules, in the sense that they know what tasks to perform with these modules. However the performability of these tasks is represented with dependencies. However according to Axiom 1 the dependencies of a module are always the same. So the knowledge of a module from a user implies, through the dependency graph, the knowledge of its dependencies as well, and therefore the knowledge of performing various tasks with the module. We discuss the consequences of not making this assumption later in Section 3.2.5.

Figure 3.6 shows the dependency graph for two digital objects, the first being a document in PDF format (`handbook.pdf`) and the second a file in FITS format (`mars.fits`). Moreover two DC profiles over these modules are defined. The first (in blue) is a DC profile for the community of astronomers and contains the modules $T(u_1)$ ={`FITS Documentation`, `FITS S/W`, `FITS Dictionary`} and the other (in red) is defined for the community of ordinary users and contains the modules $T(u_2)$ ={`PDF Reader`, `XML Viewer`}. So for example every astronomer (every user having DC profile u_1) understands the module `FITS S/W`, in the sense that she knows how to use this software application.

The knowledge of a module from a user means that the user is able to perform all the tasks with the module. Therefore she can understand its dependencies. For example astronomers know the module `FITS S/W` denoting the software application for FITS files. They know how to run this application, so they know the dependency `FITS S/W` > `JVM`. So if we want to find all the modules that are understandable from the users of a DC profile, then we must resolve all the direct and indirect dependencies of their known modules ($T(u)$). For example the modules that are understandable from astronomers are {`FITS Documentation`, `FITS S/W`, `FITS Dictionary`, `PDF Reader, JVM, XML Viewer`}, however their DC profile is a subset of the above modules.

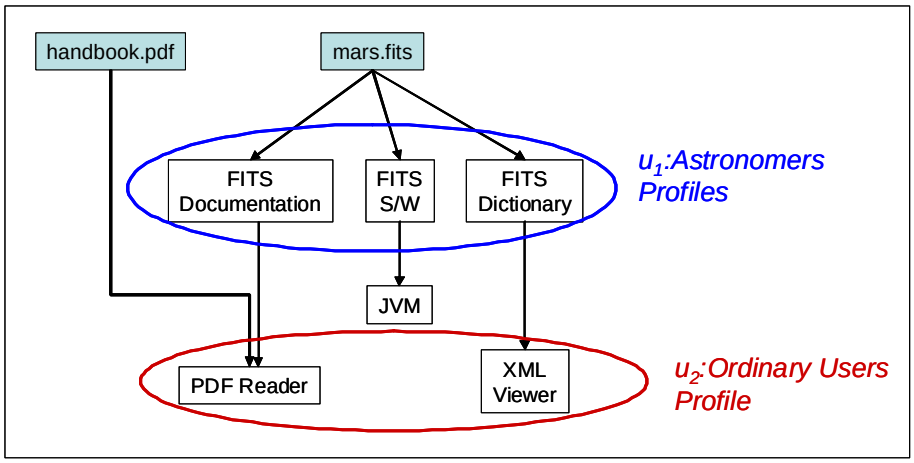

Figure 3.6: DC Profiles Example

This approach allows us to reduce the size of DC profiles by keeping only the maximal modules (maximal with respect to the dependency relation) of every DC profile. Therefore we can remove from the DC profile modules whose knowledge is implied from the knowledge of an "upper" module and the dependency graph. For example if the astronomers profile contained also the module `JVM` then we could safely remove it since its knowledge from astronomers is guaranteed from the knowledge of `FITS S/W`.

3.2.3 Intelligibility-related Preservation Services

3.2.3.1 Deciding Intelligibility

Assume the following scenario: There are two users u_1, u_2 who want to reproduce the music of an mp3 file. The user u_1 successfully reproduces the file while u_2 does not recognize it and requests its dependencies. The dependency that has been recorded is that the mp3 file depends on the availability of a mp3-compliant player, say `Winamp` (regarding reproducibility). The user is informed about this dependency and installs the application. However she claims that the file cannot be reproduced. This occurs because the application in turn has some other dependencies which are not available to the user, i.e. `Winamp` > `Lame_mp3`. After the installation of this dependency, user u_2 is able to reproduce the file. So the ability to perform a task depends on the knowledge that is assumed to be known, as well as on the dependencies of the module. However these dependencies might not be enough to ensure the performability of the task (as shown above) and therefore we must resolve the dependencies transitively.

In general we define intelligibility as the ability to perform several tasks with a module. In order to decide the intelligibility of a module from a user we must find all the necessary modules by traversing the dependency graph and recording the encountered modules, and then compare them with the modules of the DC profile of a user. However the disjunctive nature of dependencies complicates this decision. Disjunctive dependencies are used to denote the different ways to perform a task. This is translated in several paths at the dependency graph and we must find at least one such path that is intelligible by a user. The problem becomes even more complicated due to the properties (e.g. transitivity) that dependencies may have (which can be specified using rules as described in Section 3.2.1.1). On the other hand the path that is obtained from the dependency graph for the performability of a task when dependencies are conjunctive is always unique. Below we describe the different approaches that are used to decide the intelligibility of an object for disjunctive and conjunctive dependencies.

Conjunctive Dependencies

If dependencies are interpreted conjunctively then this means that the module requires the existence of all its dependencies for the performability of a task. To this end we must resolve all the dependencies transitively, since a module t will depend on t', this in turn will depend on t'' etc. Consequently we introduce the notions of required modules and closure.

- The set of modules that a module t **requires** in order to be intelligible is the set $Nr^+(t) = \{t' | t >^+ t'\}$.
- The **closure** of a module t, is the set of the required modules plus module t. $Nr^*(t) = \{t\} \cup Nr^+(t)$.

The notation $>^+$ is used to denote that we resolve dependencies transitively. So in order to retrieve the set $Nr^+(t)$ we must traverse the dependency graph starting from module t and recording every module we encounter. For example the set Nr^+(`mars.fits`) in Figure 3.6 will contain the modules {`FITS Documentation`, `FITS S/W`, `FITS Dictionary`, `PDF Reader, JVM, XML Viewer`}. This is the full set of modules that are required for making the module `mars.fits` intelligible.

In order to decide the intelligibility of a module t with respect to DC profile of a user u we must examine if the user of that profile knows the modules that are needed for the intelligibility of t. To this end we define that:

Definition 2 *A module t is intelligible by a user u, having DC profile $T(u)$ iff its required modules are intelligible from the user, formally $Nr^+(t) \subseteq Nr^*(T(u))$*

Recall that according to Axiom 1, the users having profile $T(u)$, will understand the modules contained in the profile, as well as all their required modules. In other words they can understand the set $Nr^*(T(u))$.

For example the module `mars.fits` is intelligible by astronomers (users having the DC profile u_1) since they already understand all the modules that are required for making `mars.fits` intelligible. Also the module `handbook.pdf` is intelligible by users having DC profiles u_1, u_2.
$Nr^+(\texttt{mars.fits}) \subseteq Nr^*(T(u_1))$
$Nr^+(\texttt{handbook.pdf}) \subseteq Nr^*(T(u_2))$
$Nr^+(\texttt{handbook.pdf}) \subseteq Nr^*(T(u_1))$
$Nr^+(\texttt{mars.fits}) \nsubseteq Nr^*(T(u_2))$

Disjunctive Dependencies

However the above approach will give incorrect results when the dependencies are disjunctive. Disjunctive dependencies are used to denote the different ways to perform a task which leads to multiple paths of required modules for the performability of the task. Therefore the set of required modules in this case will not be unique. Consider for example the dependencies that are shown in Figure 3.7. The dependencies in this example are interpreted disjunctively. So the module o depends on either the module t_1, or t_2. In this case we have two possible sets for $Nr^+(o)$; $Nr^+(o) = \{t_1, t_3\}$ and $Nr^+(o) = \{t_2, t_4\}$.

In the case of disjunctive dependencies we model modules as facts, and tasks and their dependencies using rules (Section 3.2.1.1). The dependency graph, like the occasion of the conjunctive dependencies, will be the same for different users. It will contain the tasks, the dependencies and the taxonomies of tasks and modules. Different users will have different modules that are available to them. For example Figure 3.8 shows a proposed architecture of a system based on facts and rules. In the upper layer the boxes contain information available to all the users regarding the tasks, their dependencies and any taxonomies of tasks and modules. The lower layer contains the modules that are available to each user. Every box corresponds to a user i.e. *James* has a file `HelloWorld.java` in his laptop and also has installed the applications `Notepad` and `VI`. If a user wants to perform a task with a module then she can use the facts of

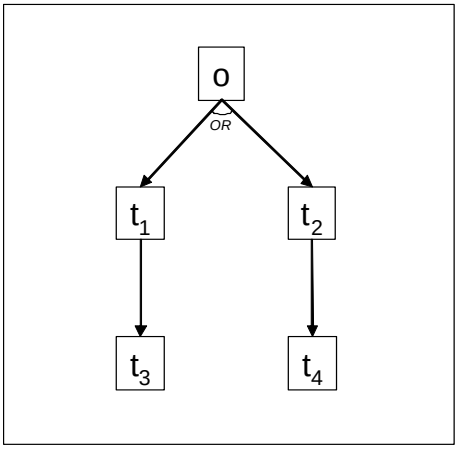

Figure 3.7: The disjunctive dependencies of a digital object *o*

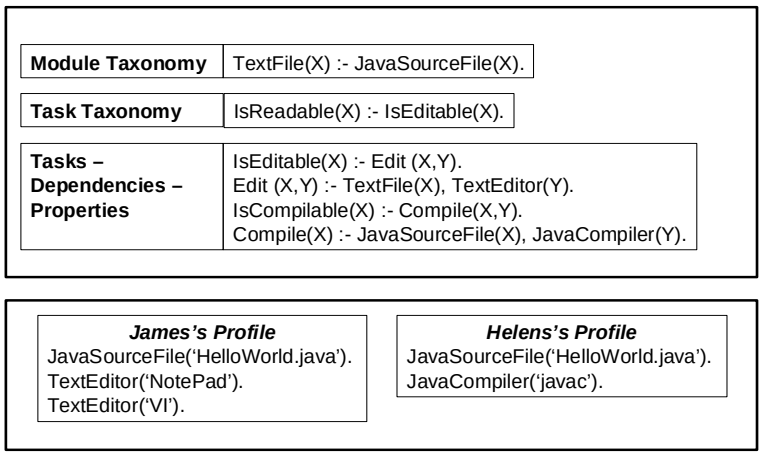

Figure 3.8: A Partitioning of facts and rules

her box and also exploit the rules from the the boxes of the upper layer.

Suppose now that *James* wants to edit the module `HelloWorld.java`. To this end he must use a set of rules allowing him to deduce if he is able to perform this task. So the problem of deciding the intelligibility of a module relies on Datalog query answering over the set of modules that are available to the user. More specifically we send a Datalog query regarding the task we want to perform with a module and check if the answer of the query contains that module. If the module is not in the answer then the task cannot be performed and some additional modules are required. For example in order to check the editability of `HelloWorld.java` *James* will send the Datalog query `IsEditable(X)`. The answer will contain the above module denoting that *James*

can successfully perform the task. In fact he can edit it using any of the two different text editors that are available to him. Suppose now that he wants to compile this file and therefore sends a query of the form `IsCompilable(X)`. However the answer of this query is empty since *James* does not have a compiler for performing the task. Similarly *Helen* can only compile `HelloWorld.java` and not edit it.

3.2.3.2 Discovering Intelligibility Gaps

If a module is not intelligible from the users of a DC profile, then we have an *Intelligibility Gap* and we must provide extra modules to the users of that community in order to understand the module. More specifically the intelligibility gap will contain only those modules that are required for the user to understand the module. This policy enables the selection of only the missing modules instead of the selecting all the required modules and hence avoiding redundancies. However the disjunctive nature of dependencies makes the computation of intelligibility gap complicated because of the multiple paths that exist in the dependency graph. So we distinguish the cases of conjunctive and disjunctive dependencies for the computation of intelligibility gap.

Conjunctive Dependencies

If the dependencies have conjunctive semantics then a task can be performed only if all its required modules are available. If any of these modules is missing then the module is not intelligible from the user. So if a module t is not intelligible by a user u, then intelligibility gap is defined as follows:

Definition 3 *The intelligibility gap between a module t and a user u with DC profile $T(u)$ is defined as the smallest set of extra modules that are required to make it intelligible. Formally* $Gap(t, u) = Nr^{+}(t) \setminus Nr^{*}(T(u))$

For example in Figure 3.6 the module `mars.fits` is not intelligible from ordinary users (users with DC profile u_2). Therefore its intelligibility gap will be:
$Gap($`mars.fits`$, u_2) = \{$`FITS Documentation, FITS SW, FITS Dictionary, JVM`$\}$
Notice that the modules that are already known from u_2 (`PDF Reader, XML Viewer`) are not included in the gap. If a module is already intelligible from a user, the intelligibility gap will be an empty set denoting that there is no need to provide extra information about the module in order to be understandable from the user. (i.e. $Gap($`mars.fits`$, u_1) = \emptyset$).

Suppose now that we have a module and we want to perform a task with it, i.e. read it. If we fail to read it, there will be an intelligibility gap, whose modules might be more than those we are in need of. This happens because the above definition of intelligibility gap does not consider the different interpretations of dependencies. Consequently we enrich the above notion with dependency types in order to retrieve the exact set of modules that are missing for the performability of A specific task.

As a motivating example suppose that we have the file `HelloWorld.java`, shown in figure 3.9. This file depends on the modules `javac` and `Notepad` representing the tasks of compiling and editing it correspondingly. Furthermore a DC profile u is specified containing the module `Notepad`. Suppose that a user, having DC Profile u, wants to edit the file `HelloWorld.java`. The editability of this file requires only the module `Notepad`. However if the user requests for the intelligibility gap ($Gap($`HelloWorld.java`$, u)$) it will contain the module `javac` even if it is not required for the editability of the module.

So we must traverse the dependencies of a specific type. However dependency types are organized hierarchically and therefore we must also include all the subtypes of the given type. Additionally the user might request the dependencies for the performability of more than one tasks, by issuing several dependency types. Given a set of dependency types W, we define:

$$t >_W t' \text{ iff (a) } t > t' \text{ and (b) } types(t > t') \cap W^* \neq \emptyset$$

where $types(t > t')$ is used to denote the types of the given dependency and W^* is the set of all possible subtypes of the given set of types, $W^* = \cup_{d \in W}(\{d\} \cup \{d' \mid d \sqsubseteq d'\})$.
So we require that at least one type of the dependency $t > t'$ must exist in the set W or it must be a subtype of one type in W. This is because (a) a dependency might be used for the performability of more than one tasks (and therefore is assigned more than one types), but we are only interested in performing a single task, and (b) we want to perform all the tasks that are denoted by W, so we must get all the dependencies that are due to any of these tasks, which is actually the union of all the dependencies for every task denoted in W. The intelligibility gap between a module t and a user u with respect to a set of dependency types W is defined as

$$Gap(t, u, W) = \{\ t' \mid t >^+_W t'\} \setminus Nr^*(T(u))$$

Below we describe the intelligibility gaps between the module `HelloWorld.java` and the DC profile u for the compilation and editability of the file.
$Gap($`HelloWorld.java`$, u, \{\_edit\}) = \emptyset$

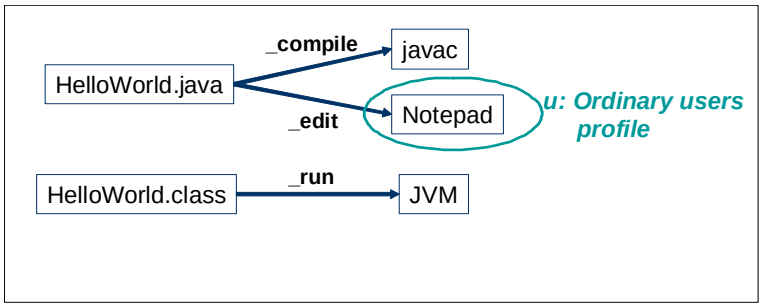

Figure 3.9: Dependency Types and Intelligibility Gap

$Gap(\texttt{HelloWorld.java}, u, \{\_compile\}) = \{\texttt{javac}\}$

$Gap(\texttt{HelloWorld.java}, u, \{\_edit, \_compile\}) = \{\texttt{javac}\}$

Disjunctive Dependencies

If the dependencies are interpreted disjunctively and a task cannot be carried out, there can be more than one ways to compute the intelligibility gap. This is due to the several paths that can be followed at the dependency graph for the performability of the task. To this end we must find the possible explanations (the possible modules) whose availability would entail a consequence (the performability of the task). For example assume that *James*, from Figure 3.8, wants to compile the file `HelloWorld.java`. Since he cannot compile it, he wants to know how to do it. Therefore we must find and deliver him the possible facts that will allow him to perform the task.

In order to find the possible explanations of a consequence we can use abduction [23, 8, 12]. Abductive reasoning allows inferring an atom as an explanation of a given consequence. There are several models and formalizations of abduction. Below we describe how the intelligibility gap can be computed using logic-based abduction. Logic-based abduction can be described as follows: Given a logical theory T formalizing a particular application domain, a set M of predicates describing some manifestations (observations or symptoms), and a set H of predicates containing possible individual hypotheses, find an explanation for M, that is, a suitable set $S \subseteq H$ such that $S \cup T$ is consistent and logically entails M. Consider for example that *James* (from Figure 3.8) wants to compile the file `HelloWorld.java`. He cannot compile it and therefore he wants to find the possible ways of compiling it. In this case the set T would contain all the tasks and their dependencies, as well as the taxonomies of modules and tasks

(the upper part of Figure 3.8). The set M would contain the task that cannot be performed (i.e. `IsCompilable(X)`). Finally the set H would contain all the modules that exist and are possible explanations for the performability of the task (i.e. all modules in the lower part of Figure 3.8). Then `JavaCompiler('javac')` will be an abductive explanation, denoting the necessity of this module for the compilability of the file. Additionally if there are more than one possible explanations then logic-based abduction would result all of them. Finally one can define criteria for picking an explanation as "the best explanation" rather than returning all of them.

3.2.3.3 Profile-Aware Packages

The availability of dependencies and community profiles allows deriving *packages*, either for archiving or for dissemination, that are *profile-aware*. For instance OAIS [20] distinguishes packages to AIPs (Archive Information Packages), which are Information Packages consisting of Content Information and the associated Preservation Description Information (PDI) and DIPs (Dissemination Information Packages), that are derived from one or more AIPs as a response to a request of an OAIS. The availability of explicitly stated dependencies and community profiles, enables the derivation of packages that contain exactly those dependencies that are needed so that the packages are intelligible by a particular DC profile and are redundancy-free. For example in Figure 3.6 if we want to preserve the file `Mars.fits` for astronomers (users with DC profile u_1) then we do not have to record any dependencies since the file is already intelligible by the community. If on the other hand we want to preserve this module for the community of ordinary users (users with DC profile u_2), then we must also record the modules that are required for this community in order to understand the module.

Definition 4 *The* (dissemination or archiving) package *of a module t with respect to a user or community u, denoted by, $Pack(t, u)$, is defined as:*
$Pack(t, u) = (t, Gap(t, u))$

Figure 3.10 shows the dependencies of a digital object o_1 and three DC profiles. The dependencies in the example are conjunctive. The packages for each different DC profile are shown below:
$Pack(o_1, DC_1) = (o_1, \{t_1, t_3\})$
$Pack(o_1, DC_2) = (o_1, \{t_1, t_2, t_4\})$
$Pack(o_1, DC_3) = (o_1, \{t_1, t_2, t_3, t_4, t_5, t_6\})$

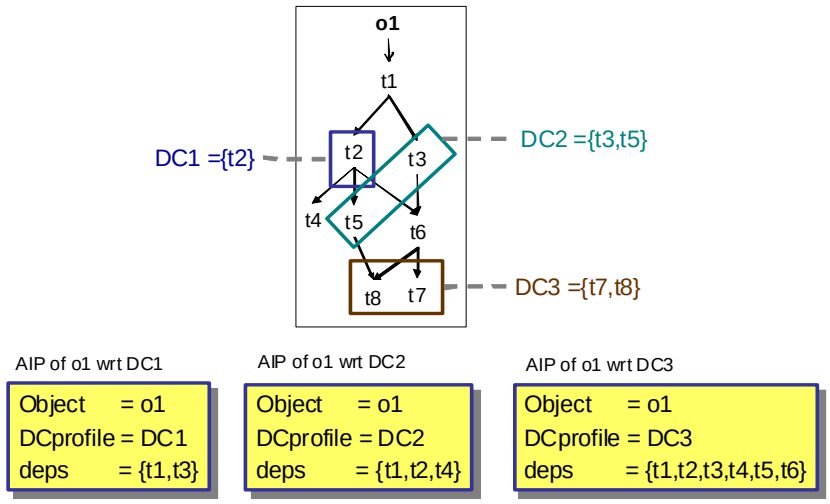

Figure 3.10: Exploiting DC Profiles for defining the "right" AIPs

We have to note at this point that there is not any qualitative difference between DIPs and AIPs from our perspective. The only difference is that AIPs are formed with respect to the profile decided for the archive, which we can reasonably assume that it is usually richer than user profiles. For example in Figure 3.10 three different AIPs for module o_1 are shown for three different DC Profiles. The DIPs of module o_1 for the profiles DC_1, DC_2 and DC_3 are actually the corresponding AIPs without the line that indicates the profile of each package.

Additionally Community knowledge evolves and consequently DC profiles may evolve over time. In that case we can reconstruct the AIPs according to the latest DC profiles. Such an example is illustrated in Figure 3.11. The left part of the figure shows a DC profile over a dependency graph and at the right part it is a newer, enriched version of the profile. As a consequence the new AIP will be smaller than the original version.

3.2.3.4 Dependency Management and Ingestion Quality Control

The notion intelligibility gap allows reducing the amount of dependencies that have to be archived/delivered on the basis DC profiles. Another aspect of the problem concerns the ingestion of information. Specifically, one rising question is whether we could provide a mechanism

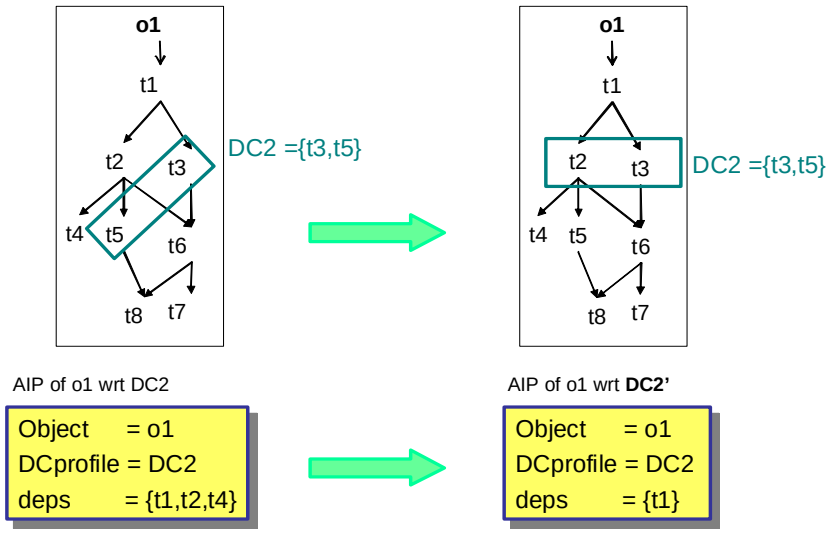

Figure 3.11: Revising AIPs after DC profile changes

(during ingestion or curation) for identifying the representation information that is required or missing. This requirement can be tackled in several ways: (a) we require each module to be classified (directly or indirectly) to a particular class, so we define certain facets and require classification with respect to these (furthermore some of the attributes of these classes could be mandatory), (b) we define some dependency types as mandatory and provide notification services returning all those objects which do not have any dependency of that type, (c) we require that the dependencies of the objects should (directly or indirectly) point to one of several certain profiles. Below we elaborate on policy (c).

Definition 5 *An module t is* related *with a profile u, denoted by $t \mapsto u$, if $Nr^*(t) \cap Nr^*(T(u)) \neq \emptyset$.*

This means that the direct/indirect dependencies of a module t lead to one or more elements of the profile u. At the application level, for each object t we can show all *related* and *unrelated* profiles, defined as:

$RelProf(t) = \{\ u \in U \mid t \mapsto u\}$ and

$UnRelProf(t) = \{\ u \in U \mid t \not\mapsto u\}$ respectively.

Note that $Gap(t, u)$ is empty if either t does not have any recorded dependency or if t has dependencies but they are known by the profile u. The computation of the related profiles allows the curators to distinguish these two cases ($RelProf(t) = \emptyset$ in the first and

$RelProf(t) \neq \emptyset$ in the second). If $u \in RelProf(t)$ then this is just an indication that t has been described with respect to profile u, but it does not guarantee that its description is complete with respect to that profile.

If dependencies are interpreted disjunctively, the set of related profiles can be found similarly. Specifically to identify if a module is related with a profile we must compute the intelligibility gap with respect to that profile and with respect to an empty profile (a profile that contains no facts at all), $gap1$ and $gap2$ correspondingly. Since dependencies are disjunctive there might exist more than one intelligibility gaps, so let $gap1$ and $gap2$ be the union of all possible intelligibility gaps. If the two sets contain the same modules (the same facts), i.e. $gap1 = gap2$, then the module will not be related with the profile. On the contrary if $gap1$ is a subset of $gap2$, this means that the profile contains some facts that are used to decide the intelligibility of the module and therefore the module is related with that profile. Finally we do the same for every profile that exists to find all the related and unrelated profiles.

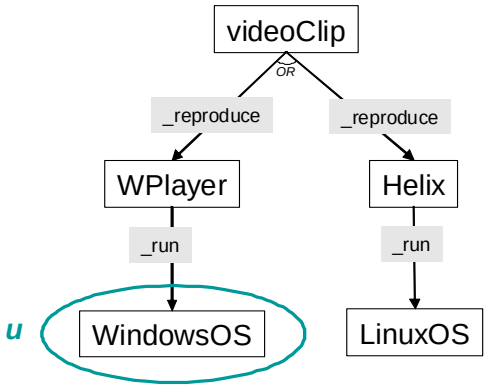

Figure 3.12: Identifying related profiles when dependencies are disjunctive

Let's clarify this issue with the following example; Figure 3.12 shows the disjunctive dependencies of a digital video file regarding the ability to reproduce it. Suppose that we want to find if that module is related with the profile u which contains the module `WindowsOS`. To this end we must compute the union of all intelligibility gaps with respect to u and with respect to an empty profile, $gap1$ and $gap2$ correspondingly. Since there are two ways to reproduce the file, there will be two intelligibility gaps whose unions will contain:

$gap1 = \{\texttt{WPlayer}, \texttt{Helix}, \texttt{LinuxOS}\}$

$gap2 = \{\texttt{WPlayer}, \texttt{WindowsOS}, \texttt{Helix}, \texttt{LinuxOS}\}$

Here $gap1 \subset gap2$ which implies that $\texttt{videoClip} \mapsto \texttt{u}$.

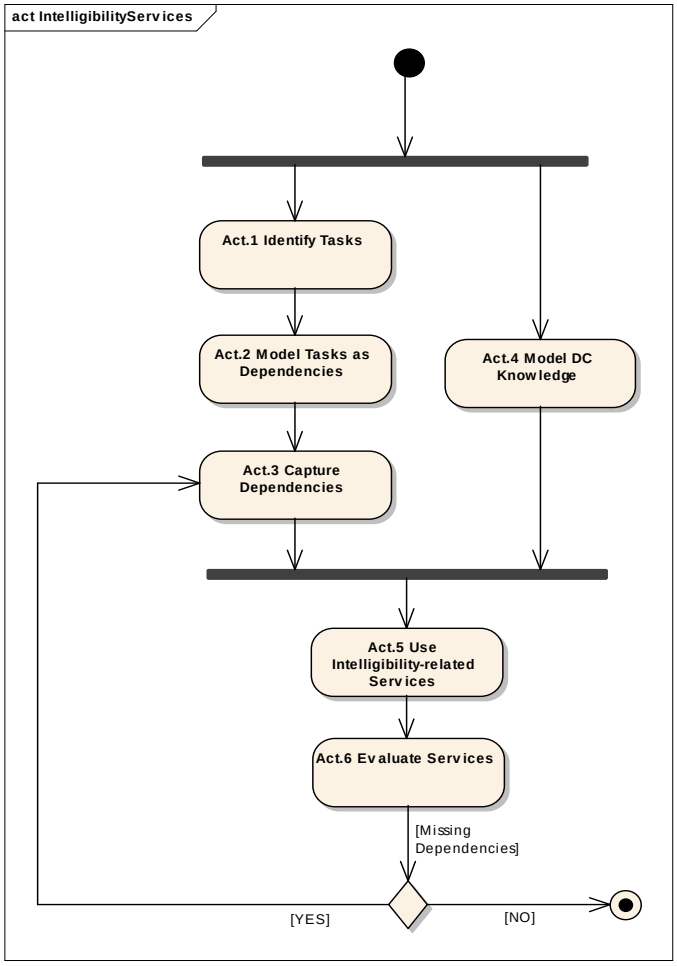

Figure 3.13: Methodological steps for exploiting intelligibility-related services

3.2.4 Methodology for Exploiting Intelligibility-related Services

Below we describe a set of activities that could be followed by one organization (or digital archivist/curator) for advancing its archive with intelligibility-related services. Figure 3.13 shows an activity diagram describing the procedure that could be followed. In brief the activities concern the identification of tasks, the capturing of dependencies of digital objects, the description of community knowledge, the exploitation of intelligibility-related services, the evaluation of the services and the curation of the repository if needed. Additionally we provide an example clarifying how an archivist can preserve a set of 2D and 3D images for the communities of DigitalImaging and Ordinary users.

Act. 1 **Identification of tasks.**

The identification of the tasks that can be performed with the modules of an archive is very important since these tasks will determine which are the dependencies of the modules. For the set of digital images the most obvious task is to render them on screen. Furthermore given that there are two different types of images we can specialize the task to the subtasks *2Drendering* and *3Drendering*.

Act. 2 **Model tasks as dependencies.**

The identified tasks of the previous activity can be modeled using dependency types. Moreover if there are tasks that can be organized hierarchically then this should be reflected to the definition of dependencies. For the tasks that were identified from the previous activity we define the following dependency types: *render render2D*, *render3D* and we would define two subtype relationships:
render2D $\sqsubseteq$ *render* and
render3D $\sqsubseteq$ *render*.
We can also determine hierarchies of modules appropriately, by defining module types, i.e. `2DImage` $\sqsubseteq$ `Image`.

Act. 3 **Capture the dependencies of digital objects.**

This can be done manually, automatically or semi-automatically. Tools like the one presented in Chapter 5 can aid this task. For example if we want to render a 2D image then we must have an `Image Viewer` and if we want to render a 3D image we must have the `3D Studio` application. So some indicative dependencies that will be recorded are:
`landscape.jpeg` $>_{render2D}$ `Image Viewer`
`Illusion.3ds` $>_{render3D}$ `3D Studio` etc.

Act. 4 **Modeling community knowledge.**

This activity enables the selection of those modules that are assumed to be known from communities of users. To this end we define two profiles; A DigitalImaging profile containing the modules {`Image Viewer`, `3D Studio`}, and an Ordinary profile containing the module {`Image Viewer`}. The former is a profile referring to users that are familiar with digital imaging technologies and the later for everyday users. The activities of modeling the community knowledge (Act.4) and capturing the dependencies of digital objects (Act. 3)

can be performed in parallel. For example this would allow the reduction of the dependencies that should be captured, i.e. we will not further analyze the dependencies of `Image Viewer` because it is already known from both profiles.

Act. 5 **Exploit the intelligibility-related services according to the needs.**
For instance the intelligibility-related services can be articulated with monitoring and notification services. For example, which modules are going to be affected if we remove the `3D Studio` from our system? These modules are:
$\{t | t >$`3D Studio`$\}$
or which are the missing modules, if an ordinary user u wants to render the 3D image `Illusion.3ds`. These modules are:
$Gap($`Illusion.3ds`$, u, \_render)$ or
$Gap($`Illusion.3ds`$, u, \_render3D)$

Act. 6 **Evaluate the services in real tasks and curate accordingly the repository**.
For instance, in case the model fails, i.e. in case the gap is empty but the consumer is unable to understand the delivered module, this is due to dependencies which have not been recorded. For example assume that an ordinary user wants to render a 3D image. To this end we deliver her the intelligibility gap which contains only the module `3D Studio`. However the user claims that the image cannot be rendered. It is because there is an additional dependency that not been recorded, i.e. the `matlib` library is required to render 3D model correctly. So a corrective action would be to add this dependency (using the corresponding activity, Act. 3). Synopsizing, empirical testing is a useful guide for defining and enriching the graph of dependencies

3.2.5 Relaxing Community Knowledge Assumptions

DC profiles are defined as sets of modules that are assumed to be known from the users of a Designated Community. According to Axiom 1 the users of a profile will also understand the dependencies of a module and therefore they will be able to perform all the tasks that can be performed. However users may know how to perform certain tasks with a module rather than performing all of them. For example assume the case of java class files. Many users that are familiar with java class files know how to run them (they know the module denoting the java

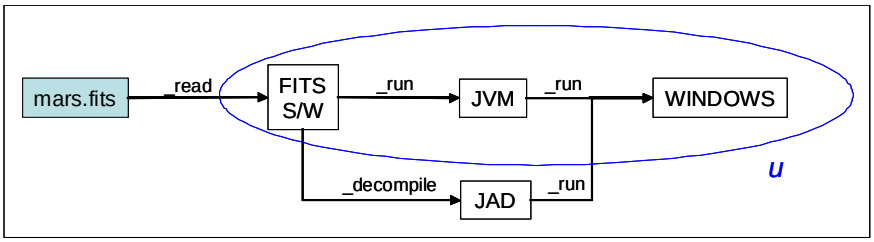

Figure 3.14: Modeling DC profiles without making any assumptions

class files and they understand its dependencies regarding its runability, i.e. the module JVM), but many of them do not know how to decompile them.

In order to capture such cases, we must define DC profiles without making any assumptions about the knowledge they convey, (as implied from Axiom 1). No assumptions about the modules of a profile means that the only modules that are understandable from the user u of the profile are those in the set $T(u)$. Additionally the only tasks that can be performed are those whose modules and their dependencies exist in the DC profile. For example Figure 3.14 shows some of the dependencies of a file in FITS format. The users of profile u know how to run the module `FITS S/W` since all the modules it requires exist in $T(u)$. However they cannot decompile it, even if they know the module `FITS S/W`, since we cannot make the assumption that a user u will also understand `JAD` and its dependencies.

However such an alteration on DC profiles would result in changing some of the definitions and intelligibility-related services that include profiles, since the set of all the modules that are understandable to the users of a profile is $T(u)$ (instead of $Nr^*(T(u))$). Therefore the way of deciding the intelligibility and computing the intelligibility gap should change as follows:
Deciding Intelligibility: $Nr^+(t) \subseteq T(u)$
Intelligibility Gap: $Nr^+(t) \setminus T(u)$
Another consequence is that we cannot reduce the size of DC profiles be keeping only the maximal elements since no assumptions about the knowledge are permitted.

In the case where dependencies are disjunctive, we do not make any assumptions about the knowledge that is assumed to be known, since the properties of various tasks and their dependencies are denoted explicitly. In this case the performability of a task is modeled using 2 intentional predicates. The first is used for denoting the task i.e.

```
IsEditable(X) :- Editable(X,Y).
```

and the second for denoting which are the dependencies of this task i.e.

```
Editable(X,Y) :- TextFile(X), TextEditor(Y).
```

DC profiles contain these modules that are available to the users (i.e. `TextEditor('NotePad').`). So in order to examine if a task can be performed with a module we rely on specific module types as they have been recorded in the dependencies of the task, i.e. in order to read a **TextFile** X then Y must be a **TextEditor**. However users may know how to perform such a task without necessarily classifying their modules to certain module types or they can perform it in a different way than the one that is recorded.

Such dependencies can be captured by enriching DC profiles with extensionally predicates (with arity greater than 2) that denotes the knowledge of a user to perform a task in a particular way, and associating these predicates with the predicates regarding the performarbility of the task. For example for the task of editing a file we will define three predicates:

```
IsEditable(X) :- Editable(X,Y).
Editable(X,Y) :- TextFile(X), TextEditor(Y).
Editable(X,Y) :- EditableBy(X,Y).
```

The predicates IsEditable and Editable are intentional while the predicate EditableBy is extensional. This means that a user who can edit a text file with a module which is not classified as a TextEditor, and wants to define this explicitly, could use the following fact in his profile

```
EditableBy('readme.txt', 'myProgr.exe').
```

3.3 Modeling and Implementation Frameworks

3.3.1 Conjunctive Dependencies using Semantic Web languages

Semantic web languages can be exploited for the creation of a standard format for input, output and exchange of information regarding modules, dependencies and DC profiles. To this end we created an ontology (expressed in RDFS). Figure 3.15 sketches the backbone of this ontology. We shall hereafter refer to this ontology with the name `COD` (Core Ontology for representing Dependencies). This ontology contains only the notion of DC Profile and Module and consists of only two RDF Classes and five RDF Properties (and it does not define any module or dependency type). It can be used as a standard format for representing and exchanging information regarding modules, dependencies and DC profiles. Moreover it can guide the specification of the message types between the software components of a preservation

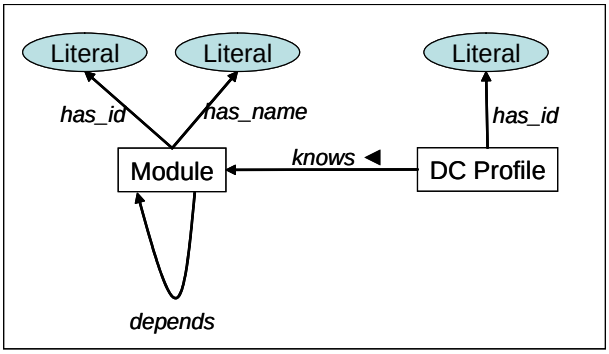

Figure 3.15: The Core Ontology for representing Dependencies (COD)

information system.

The adoption of Semantic Web languages also allows the specialization of COD ontology according to the needs. Suppose for example we want to preserve the tasks that can be performed with the objects of Figure 3.9, specifically _edit, _compile, _run. These tasks are represented as dependency types, or in other words as special forms of dependencies. To this end we propose specializing the dependency relation, by exploiting "subPropertyOf" of RDF/S. This approach is beneficial since: (a) a digital preservation system whose intelligibility services are based on COD will continue to function properly, even if its data layer instantiates specializations of COD and (b) the set of tasks that can be performed cannot be assumed a priori, so the ability to extend COD with more dependency types offers extra flexibility.

Additionally COD could be related with other ontologies that may have narrower, wider, overlapping or orthogonal scope. For example someone could define that the dependency relation of COD corresponds to a certain relationship, or path of relationships, over an existing conceptual model (or ontology). For example the data dependencies that are used to capture the derivation of a data product in order to preserve its understandability, could be modeled according to OPM ontology [32] using *wasDerivedFrom* relationships, or according to the CIDOC CRM Ontology (Chapter 4) using paths of the form:
$S22$ *was derivative created by* $\rightarrow$ `C3 Formal Derivation` $\rightarrow$ $S21$ *used as derivation source.*
The adoption of an additional ontology allows capturing metadata that cannot be captured only with COD. For example, assume that we want to use COD for expressing dependencies but we also want to capture provenance information according to CIDOC CRM Digital ontology. To gain this functionality we should merge appropriately these ontologies. For instance if we want

every `Module` to be considered as a `C1 Digital Object`, then we would define that `Module` is a `subClassOf C1 Digital Object`. Alternatively one could define every instance of `Module` also as an instance of `C1 Digital Object`. The ability of multiple classification and inheritance of Semantic Web languages gives this flexibility. The upper part of Figure 3.16 shows a portion of the merged ontologies. Yellow rectangles represent classes according to CIDOC CRM Digital ontology and the blue ones describe classes from `COD` ontology. Thick arrows represent `subclassOf` relationships between classes and simple labeled arrows represent properties. The lower part of the figure demonstrates how information about the dependencies and the provenance of digital files can be described. Notice that the modules `HelloWorld.java` and `javac` are connected in two ways: (a) through the *_compile* dependency, (b) through the "provenance path"
`Module("HelloWorld.java")` $\rightarrow S21$ *was derivation source for* $\rightarrow$
`C3 Formal Derivation ("Compilation")` $\rightarrow P16$ *used specific object* $\rightarrow$ `Module("javac")`[1]
Note that an implementation policy could be to represent explicitly only (b), while (a) could be deduced by an appropriate query. Furthermore the derivation history of modules (i.e. `HelloWorld.class` was derived from `HelloWorld.java` using the module `javac` with specific parameters etc.) can be retrieved by querying the instances appropriately. More information about modeling provenance and query templates can be found at Chapter 4

3.3.2 Implementation Approaches for Disjunctive Dependencies

The model based on disjunctive dependencies is founded using Horn rules. Therefore we could use a rule language for implementing it. Below we describe three implementations approaches; Using Prolog, SWRL, and a DBMS that supports recursion.

Prolog is a declarative logic programming language, where a program is a set of Horn clauses describing the data and the relations between them. A computation is initiated by running a query over these relations. So the proposed approach can be straightforwardly expressed in Prolog. Furthermore and regarding abduction there are several approaches that either extend Prolog [7] or augment it [6] and propose a new Programming Language.

The **Semantic Web Rule Language (SWRL)** [17] is a combination of OWL DL and OWL Lite [10] with the Unary/Binary Datalog RuleML[2]. SWRL provides the ability to write

[1]The property *S21 was derivation source for* that is used in the "provenance path" is a reverse property of the property *S21 used as derivation source* that is shown in Figure 3.16. These properties have the same interpretation but inverse domain and range.
[2]http://ruleml.org

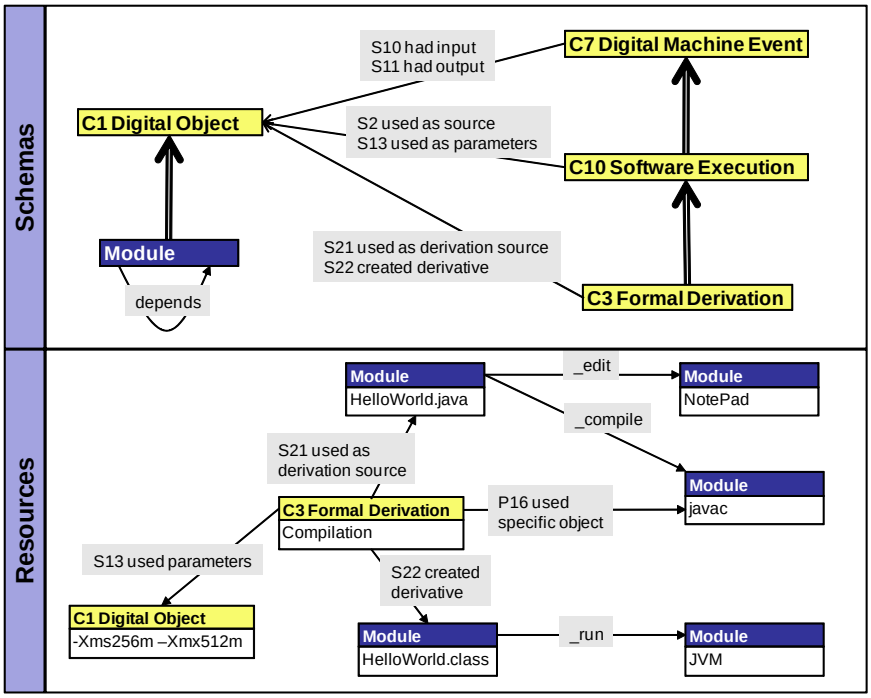

Figure 3.16: Extending COD for capturing provenance

Horn-like rules expressed in terms of OWL concepts to infer new knowledge from existing OWL KB. For instance, each type predicate can be expressed as a class. Each profile can be expressed as an OWL class whose instances are the modules available to that profile (we exploit the multiple classification of SW languages). Module type hierarchies can be expressed through *subclassOf* relationships between the corresponding classes. All rules regarding performability and the hierarchical organization of tasks can be expressed as SWRL rules.

In a **DBMS**-approach all facts can be stored in a relational database, while *Recursive SQL* can be used for expressing the rules. Specifically, each type predicate can be expressed as a relational table with tuples the modules of that type. Each profile can be expressed as an additional relational table, whose tuples will be the modules known by that profile. All rules regarding task performability, hierarchical organosis of tasks, and the module type hierarchies, can be expressed as datalog queries. Note that there are many commercial SQL servers that support the SQL:1999 syntax regarding recursive SQL (e.g. Microsoft SQL Server 2005, Oracle 9i, IBM DB2).

The following table (Table 3.1) synopsizes the various implementation approaches and describes how the elements of the model can be implemented. The table does not contain any

What	DB-approach	Semantic Web-approach
ModuleType predicates	relational table	class
Facts regarding Modules (and their types)	tuples	class instances
DC Profile	relational table	class
DC Profiles Contents	tuples	class instances
Task predicates	IDB predicates	predicates appearing in rules
Task Type Hierarchy	datalog rules, or isa if an ORDBMS is used	*subclassOf*
Performability	datalog queries (recursive SQL)	rules

Table 3.1: Implementation Approaches for Disjunctive Dependencies

information about the Prolog approach since the terminology we used for founding the model (Section 3.2.1.1) is the same with Prolog, i.e. Facts about modules are represented as facts in Prolog, rules that are used for dependencies are also rules in Prolog etc.

3.4 Implementation - `GapMgr` Tool

In this section we describe the implementation of the dependency management services based on the model we described in Section 3.2. Specifically we define the query and update services for modules, dependencies and profiles. The dependencies in the model are defined as transitive and are interpreted conjunctively. We introduce the architecture and the functionalities of a dependency management tool, `GapMgr`. Additionally we evaluate the services using several synthetic data sets and report the application results from CASPAR [1] project.

3.4.1 Intelligibility (Dependency) Management Services

The basic intelligibility-related services include the identification of dependencies between modules, the decision about the intelligibility of modules by profiles and the computation of intelligibility gaps. Since modules and dependencies can be typed we also must include them in the specification of the services. In addition we foresee the need to include several profiles for the decision of intelligibility since the knowledge of a user may be expressed using various profiles. For example a user that knows how to program in Java and also knows the concepts

and software used by astronomers (recall FITS files), will be associated with two Profiles; the `JavaProgrammersProfile` and `AstronomersProfile`. The basic query services are defined in the following table (Table 3.2).

Return Value	Service	Definition
Boolean	depends(Module t, Module t', DepTypes[] W)	$t >^{?}_{W} t'$
Module[]	RequiredModules(Module t, DepTypes[] W)	$Nr^{+}_{W}(t) = \{ t' \mid t >^{+}_{W} t'\}$
Module[]	Closure(Module[] S, DepTypes[] W)	$Nr^{*}_{W} = \cup_{t \in S}(\{t\} \cup Nr^{+}_{W}(t))$
Module[]	KnownModules(Profile[] U)	$Nr^{*}(T(U)) = \cup_{u \in U} Nr^{*}(T(u))$
Boolean	isIntelligible(Module t, Profile[] U, DepTypes[] W)	$Nr^{+}_{W}(t) \overset{?}{\subseteq} Nr^{*}(T(U))$
Module[]	Gap(Module t, Profile[] U, DepTypes[] W)	$Gap(t, U, W) = Nr^{+}_{W}(t) \backslash Nr^{*}(T(U))$

Table 3.2: Basic Query Services

All the services contain the notion of dependency types. When multiple dependency types are defined we require at least one of the dependency types must exist in the resolved dependencies. Of course these services also function without the provision of any dependency type. Additionally module types can be employed for filtering the results of the above modules.

Apart from issuing queries a dependency management system must also maintain its descriptions, since the set of modules, dependencies and profiles may change over time. Table 3.3 introduces some basic change services for updating the dependency graph and the modules that are known from profiles. Every service is defined by its pre and post conditions. The post-conditions describe not only the conditions that must be true in the dependency graph but also the side-effects of these services on profiles. The addition of a dependency must ensure the acyclicity of the dependency graph for the specified types. On the other hand the deletion of a dependency may "break" the defined profiles. To avoid such cases before executing $delDependency(t, t')$ we have to find those profiles that contain t or a broader module (since we are storing only the maximal elements in profiles). If U is the set of all profiles then the sought set of profiles is $\{u \in U | t \in Nr^{*}(T(u))\}$. If u is a profile in the above set, we add to it the modules $Nr(t)$. This means that for a revised profile u it will hold that $T(u) \leftarrow T(u) \cup Nr(t)$. After that we remove the dependency $t > t'$ and apply the maximality constraint to reduce the size of the affected profiles. For example the deletion of the dependency $t_{13} > t_{15}$ in figure 3.17 will cause side effects to the profiles u_1 and u_3. To curate profile u_3 we need to add to it the modules in $Nr(t_{13})$. Finally the revised profile will contain $T(u_3) = \{t_8, t_9, t_{13}, t_{15}\}$.
Similarly the removal of a module ($delModule(t)$) is decomposed in two different change operations; the removal of all dependencies that involve t, and deletion of the module from every

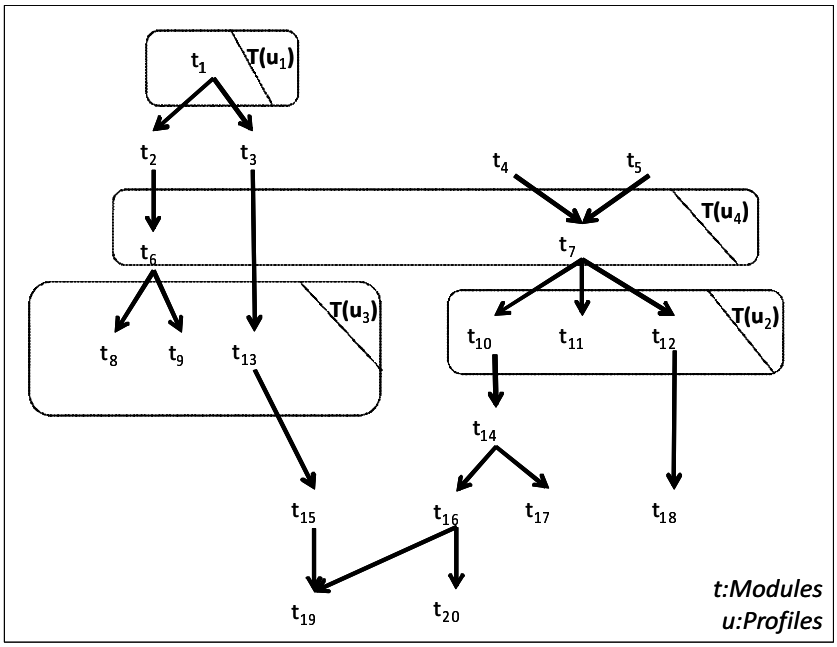

Figure 3.17: Example of Modules, Dependencies and Profiles

profile it is contained. Finally we remove t.

The operation $upgradeModule(t)$ is used to create a new module t_{new} standing as a newer version of module t. This operation is actually a shortcut which can be configured. For example, if we assume that new versions are backwards compatible, then the new module t_{new} will have the same descriptions (i.e. name, dependencies), but a revised version number. For example if we issue $upgradeModule(t_{14})$, then we will get a new module t'_{14} which also depends on t_{16} and t_{17}.

Finally there are operations for the curation of profiles ensuring the maximality constraint about profiles' known modules.

3.4.2 Implementation Settings and Experimental Evaluation

Here we detail the functionality and architecture of a tool named `GapMgr` that realizes all services described previously. It comprises:

(a) `GapMgr` API. This is a programmatic interface written in Java. It provides the concepts for defining digital objects and their dependencies, as well as the assumed knowledge on the basis of profiles.

Change Operations on Dependencies			
Ret. Value	Operation	Pre-condition	Post-Condition
Boolean	addDependency (t, t': Module, W: DepTypes[])	$t, t' \in \mathcal{T}$, $W \subseteq D$	For each $w \in W$ it holds: $t >_w t'$ and $>_w$ is acyclic
Boolean	delDependency (t,t':Module, W: DepTypes[])	$t >_W t'$	For each $w \in W$ it holds: $t \not> _w t'$ For each $u \in U$, if $t \in Nr^*(T(u))$ then $T(u) \leftarrow max_>(T(u) \cup Nr(t))$
Boolean	delModule(t: Module)	$t \in T$	All dependencies that involve t are removed (by issuing delDependency operations) For each $u \in U, T(u) \leftarrow T(u) \backslash \{t\}$ $t \notin T$
Module	upgradeModule (t: Module)	$t \in T$	a new Module t_{new} is created such that: for each $w \in W$ if $t >_w t'$ then $t_{new} >_w t'$
Change Operations on Profiles			
Ret. Value	Operation	Pre-condition	Post-Condition
void	appendKnownModules (u:Profile, S: Module[])	$S \subseteq T$	$T(u) \leftarrow max_>(T(u) \cup S)$
Boolean	delModule (u:Profile, t: Module)	$t \in Nr^*(T(u))$	$T(u) \leftarrow max_>((T(u) \cup Nr(t)) - \{t\})$

Table 3.3: Basic Change Services

(b) Two different implementations of the API. The first is a main memory implementation. The persistence layer is a plain file-system based. The second is an implementation over the SWKM (Semantic Web Knowledge Middleware)[3]. Here modules, dependencies and profiles are stored in the SWKM repository. SWKM offer a wide and scalable suite of basic services for validating, storing, querying, updating and exporting descriptive metadata expressed in RDF/S. All services are based on a common knowledge repository enabling the consistency of its contents. It offers advanced evolution services (declarative update languages, comparison services and versioning). In this setting all computations are done by sending queries (in RQL[24] and RUL[29]) to the repository through the SWKM WS client. The main memory requirements are low in this case.

(c) An end-user Web-based application developed over GWT (Google Web Toolkit)[4]. It has a modular design and it can work with both implementations of the API. Figure 3.18 shows the Use Case Diagram of this application.

The overall architecture of `GapMgr` is shown in the figure below (Figure 3.19). It depicts how the various components (`GapMgr`, SWKM, glassfish server, etc) are wired together.

Some indicative screendumps are shown in Figures 3.20 and 3.21; The former for defining dependencies and the latter for computing intelligibility-aware packages. It can be considered as a semantic registry for preservation.

[3]http://athena.ics.forth.gr:9090/SWKM/
[4]http://code.google.com/webtoolkit/

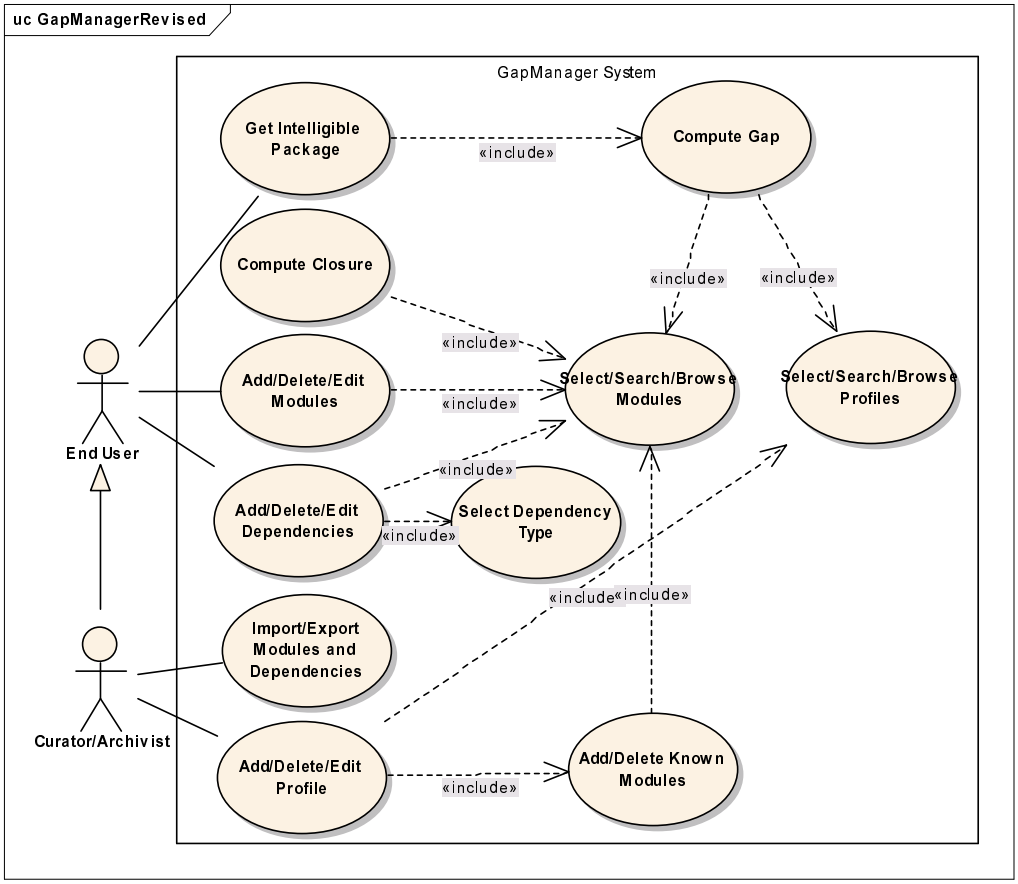

Figure 3.18: The Use Case Diagram of `GapMgr`

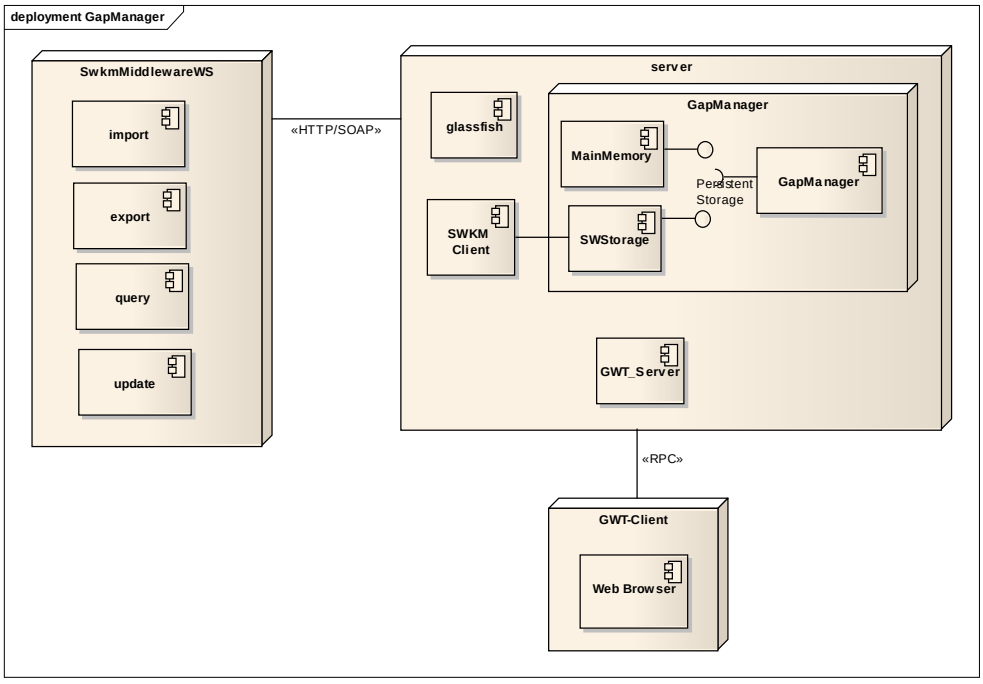

Figure 3.19: `GapMgr` Architecture

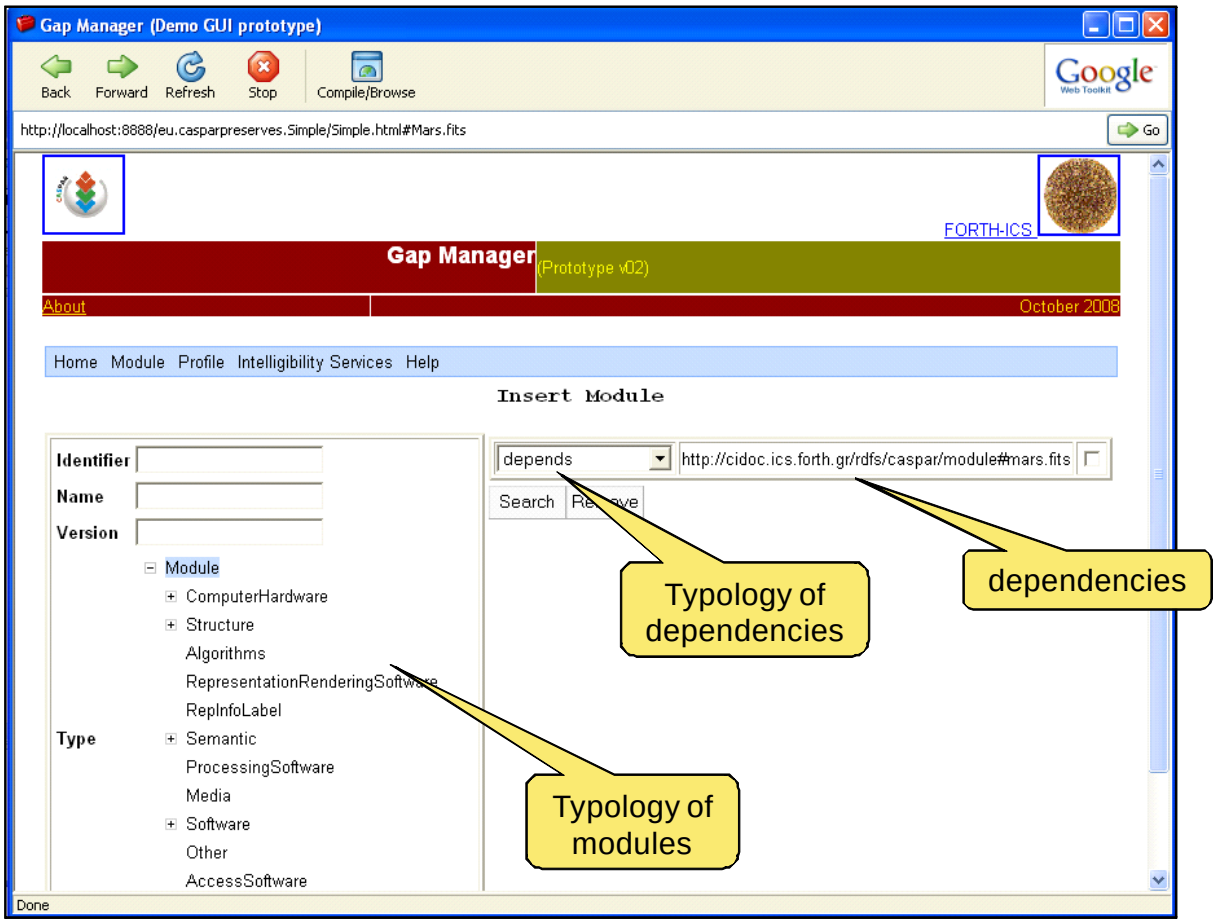

Figure 3.20: The GUI of `GapMgr`: Defining dependencies

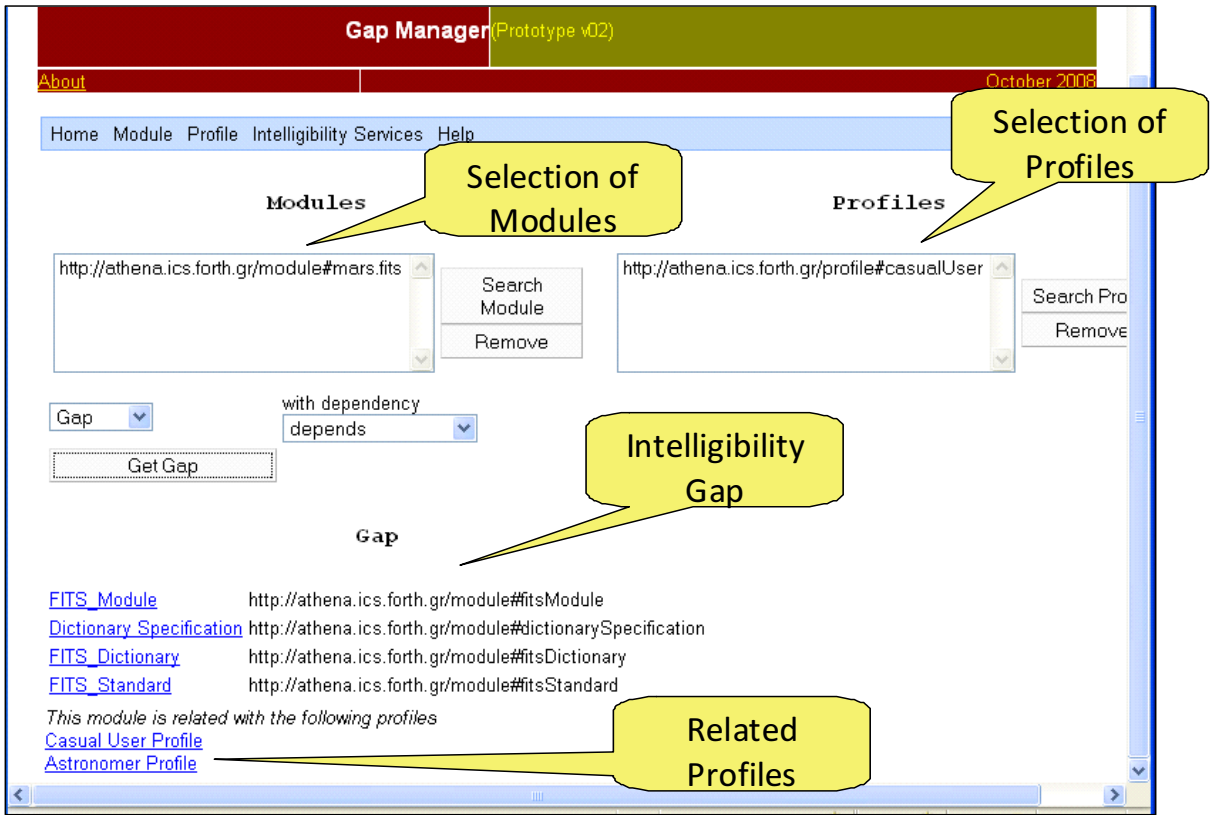

Figure 3.21: The GUI of `GapMgr`: Computation of the intelligibility gap

3.4.2.1 Experimental Evaluation

Synthetic Data Sets

In order to evaluate the efficiency of the various implementations, we have created several large synthetic data sets. Specifically we have devised a generator that takes as input two parameters: the number of modules N to be created, and the "density" of the dependency graph (two options are available *sparse* and *dense*). The generator proceeds as follows: At first it creates N modules. Then it creates random dependencies among these modules. If "sparse" then it creates $N \log N$ dependencies, otherwise (i.e. if "dense") it creates $2N \log N$ dependencies. Subsequently $\log N$ profiles are defined (we also have another option that creates N^2 profiles). To each of these profiles a random set of modules is associated in the range of $[1 \dots \sqrt{N}]$.

The average depth of the obtained dependency graph as well as the average size of the closure of a module are shown in Table 3.4. This graph is rather defined as the "worst-case scenario" since the dependency graphs for real data are usually "smaller", in the sense that the depth of a closure path will be smaller. Indeed in the subsequent section (Section 3.4.2.2), where we discuss the applicability of `GapMgr` in real datasets, the average depth of a closure is approximately 3 and its size is much smaller.

N	10^3	10^4	10^5
Average depth	9	10	10
Average closure size	261	2174	17227
Max depth	23	24	25
Max closure size	675	5930	50838

Table 3.4: Dependency Graph Depth

Observing Table 3.4 one can easily see that the average depth of the graph remains the same as the number of the contained modules increases but on the other hand the average size of its closure is increased. This means that as the number of modules grows the dependency graph becomes broader rather than deeper. This is due to the adopted generation method.

Measured Tasks (Description and Algorithms)

We measured the time (and memory) requirements for the following tasks:

- **Computation of closure**

  (i.e. $Closure(t)$ or $Closure(S)$)

We traverse the dependency graph and collect all dependent modules starting from t or S. The complexity of this task depends on the number of modules but mainly on the density of the dependency graph (the more dense the graph is, the more modules the closure will contain and the more time and memory consuming this task is).

- **Computation of gaps** (i.e. $Gap(t, u)$)

  Computing the gap between one module t and one profile u is more complex than just computing the closure of a module. This is because finding the gap contains internally the process of getting the closures of all modules that a profile contains. The complexity of this task depends on the density of the dependency graph and on the number of modules that the profiles have.

- **Deciding intelligibility** (i.e. $isIntelligible(t, u)$)

  To decide whether a module t is intelligible by one profile u we have to compute the closure of all modules that a specified profile knows. If the module t or its direct dependencies exist in the closure set, then this module is intelligible by the profile.

- **Deciding dependency** (i.e. $depends(t, t')$)

  To decide whether a module t depends on another module t' we have to check whether there is a direct or indirect dependency between them. This is true if t' is member of the closure of t (i.e. if $t' \in Nr^+(t)$). Instead of computing the entire closure of t, we can compute it gradually (i.e. traverse the dependency graph starting from t) so that to be able terminate whenever t' is going to be added to the closure. It follows that this task is faster than the computation of closure.

- **Adding a dependency**

  Before adding a dependency $t > t'$ we have to guarantee that this addition will not create a cycle. To this end we first check whether $t' >^+ t$ and if this holds we reject the request. Therefore the cost of this task is roughly equal to the cost of $depends(t, t')$.

- **Deletion of module**

  Before deleting a module we must first remove all the dependencies where t is involved. We also have to "curate" the profiles, i.e. find the profiles u where $t \in Nr^*(T(u))$ and replace that element with the modules $Nr(t)$. Finally we delete module t.

- **Module upgrade**

  A new module is created with the same dependencies with the specified module. Specifically the newly created module has exactly the same direct dependencies ($Nr(t') = Nr(t)$) with the specified one.

Evaluation Results

Table 3.5 shows the time measurements for the computation of various tasks for different numbers of modules using the main memory API of `GapMgr`. This approach stores modules, dependencies and profiles in main memory.

Services	Time(ms)		
	$N = 10^3$	$N = 10^4$	$N = 10^5$
getClosure(t)	2	51	820
depends(t,t')	1	33	426
addDependency(t,t')	1	33	411
getDirectDependencies(t)	0	0	0
getDirectDependents(t)	0	0	0
isIntelligible(t,u)	8	212	4631
deleteModule(t)	0	1	1
Gap(t,u)	13	223	5412
upgradeModule(t)	1	2	2

Table 3.5: `GapMgr` Evaluation using Main Memory API

If the number of modules is low (i.e. 1000) the tasks are performed very fast. As the total number of modules increases the total cost also increases. We can distinguish the various tasks to some broad categories: (1) Single simple query tasks, (2) Multiple simple query tasks, (3) Single complex query tasks, and (4) Multiple complex query tasks.

Single simple query tasks are those that can be dispatched using only a simple query to get their results. These tasks require the minimum time to execute. Take for example the tasks *getDirectDependencies(t)* and *getDirectDependents(t)* that take constant time even when the total number of modules is quite big. Multiple simple query tasks consist of subtasks that fall in the previous category. *deleteModule(t)* is an example of such a task since it internally uses other subtasks as we have already seen (i.e. *delDependency*).

Complex query tasks on the other hand are those whose execution is more time-demanding. To dispatch a complex query we need to traverse the dependency graph (rather than just query

the graph). The extra overhead of these tasks strongly depend on the depth and size of the dependency graph. For example a single complex query task is the computation of the closure, *getClosure(t)* which leads at resolving the dependencies between modules transitively for the whole depth of the graph. Other tasks categorized here are *depends(t,t')*, *addDependency(t,t')*. Finally multiple complex query tasks are those that contain multiple calls to tasks of the previous category. For example the gap computation (*gap(t,u)*) is such a task, since it consists of the computation of the closure of module t and the closure of every module of u. *isIntelligible(t,u)* is categorized here since it is similar to gap computation.

In the main memory implementation of the API, all modules, dependencies and profiles are loaded in main memory. This approach has proved efficient, however if the volume of data increases then they may not fit in memory. The implementation over SWKM overcomes such limitations, as all data are stored in the SWKM repository and all services are implemented by appropriate calls to the declarative query and update language (RQL and RUL respectively). However, and regarding efficiency, this approach has the overhead of parsing the results of the queries which are delivered in RDF/XML format. For big results, the delay is unacceptably long. For instance, the execution of a query whose answer comprises 2300 modules takes about 5 sec to compute and get (through the Web Service), while the results parsing takes around 50 sec. The problem could be alleviated by extending the implementation of RQL so that to support result sets and cursors. Moreover a solution to speedup queries, is the adoption of grid computing approaches (or computer clusters) that rely on combining many computer resources to process large datasets.For instance at [37] they report results of querying over 300 million tripes using Openlink Virtuoso[5].

3.4.2.2 Application Results from CASPAR

In order to see whether `COD` can be used (as it is or by extending it) for capturing several different types of modules and dependencies (from various different domains) we applied it for expressing the contents of various existing collections. So far we have successfully loaded the following:

- *File Formats and Software Products*

  PRONOM [39] is an online registry of technical information. Specifically it provides information about the file formats, software products and other technical components required

[5] http://www.openlinksw.com/virtuoso/

to support long-term access to electronic records and other digital objects of cultural, historical or business value. Currently around 850 records are listed. All have been extracted and loaded to `GapMgr`. However only a few dependencies are defined between these formats.

- *Modules and Dependencies from the CASPAR project*

  We have imported several modules along with their dependencies in the context of the CASPAR project. In addition, we defined several profiles for the various communities involved. The resulting data set contains modules from the cultural domain (UNESCO sites) and contemporary arts domain (mainly coming from CNRS, INA[6], University of Leeds, and CIANT). Furthermore, there are (more than 1200) modules exported from a Registry of formats from the scientific domain.

Additionally several web applications have been deployed and are used from several partners of the CASPAR project including including University of Leeds [7], CNRS [8], CIANT [9], European Space Agency (ESA-ESRIN) [10], British Atmospheric Data Centre [11]. Online versions of the web applications of `GapMgr` can be found at:
`http://developers.casparpreserves.eu:8080/CasparGui/`
`http://139.91.183.30:3025/GapManagerGWT_SWKM/`

3.5 Related Approaches

Dependencies are ubiquitous and dependency management is an important requirement that is subject of current research in several (old and new emerged) areas: from Software Engineering (at [46], [47], [48], [5], [4]), to Ontology Engineering (at [22], [38]). Specifically, in software engineering the various build tools (e.g. make, gnumake, nmake, jam, ant) are definitely related (they allow defining dependencies and those tasks required to be performed in order to build a software project). In ontology engineering an analogous problem is 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. Another related problem is that of schema evolution problem, i.e.

[6] Institut National de l'Audiovisuel, France
[7] http://www.leeds.ac.uk/
[8] Centre National de la Recherche Scientifique, France
[9] International Centre for Art and New Technologies, Prague, Czech Republic
[10] http://www.esa.int/esaMI/ESRIN_SITE/
[11] http://badc.nerc.ac.uk/

the problem of reflecting schema changes to the underlying instances. Actually this problem is related to the evolution of modules and dependencies.

Table 3.6 list and describes in brief a number of dependency management approaches that have been described in the literature.

Work	Modules	Objective of the Dependency	Types of Dependencies (between modules)	Reason why dependencies are recorded
[5]	Software components	To install or uninstall a composite component	Mandatory, optional, negative.	To reason on istallability, deinstallability
[15]	Software components	Achieve goals, satisfy soft goals, complete tasks, provide and consume resources	Goal, task, resource, soft goal	To aid the selection of the most appropriate component
[47]	Software components	One goal is the ability to compile/run and another is to express which component affects the behaviour of other components.	The dependencies of a component are categorized to (a) Internal (i.e. intra-component), and (b) External (inter-component). The internal ones are further categorized to (a1) implementation-based and (a2) operation-based. The external ones are further distinguished to (b1) hardware, (b2) software (i.e. required interfaces), and (b3) causal.	To support the process of evolution and testing in component-based systems.
[38]	Ontologies management	Considers the dependencies recorded in the ontology representations (reuse/extend inter-ontology relationships).	IsA, Reference	To aid the development of ontologies in particular when changes occur, i.e. to address questions of the form: if an ontology changes what should happen in the dependent ontologies?
Maven	Software components	software engineering	compile, provided, runtime, test, system, import	build, compile, and the like

Table 3.6: Dependency Management in other Domains

As we can see there is much heterogeneity on the types of modules, the objectives of the dependencies, the types of dependencies and on the dependency management services. Probably in each preservation application domain we have to model the corresponding modules and dependency types and identify the needed services. It does not seem that we could have one single modeling for all cases. This observation justifies the selection of SW languages (due to the extensibility they offer).

Below we discuss in brief some of the works that are mentioned in Table 3.6. [15] defines four types of dependencies (*goal, soft goal, task, resource*). Furthermore it describes six system properties derivable from soft goals, namely, *diversity, vulnerability, packaging, self-containment,*

uniformity, and *connectivity*. These properties are then used as criteria to select the best software architecture. Furthermore it defines light components (i.e. components that could be replaced by others serving a similar purpose) and heavy ones (i.e. components on which other components strictly depend on). For example an email server can be considered as a heavy component, while an email client as a light one.

[47] proposes a technique to analyze dependencies in large component-based systems. The main idea is to describe the dependencies of each component using XML and analyzing the influence of those dependencies in a CBS (Component-based system). In order to identify chains of dependencies, *pomsets* are used. A pomset description can express what can take place after a particular component's service call takes place.

[5] concentrates on two main attributes of the dependability problem: (a) the *success of the deployment* of a software component, and (b) the *safety* of the system. Deployment success guarantees that once a component (that consists of multiple and depending entities) is deployed it will work correctly. The main idea beyond safety is to preserve the correctness of the functionalities of the system after its deployment. To ensure this the installation process is divided into two steps, each treated as a different problem: the *installability problem* and the *post-installation effects*. This is accomplished by building a tool with formal foundations for each problem and thus ensuring the success and safety of deployment. It is based on a dependency description language where the concepts of dependency and predicate are defined. The condition of the availability of each service is expressed in first order predicate language.

[38] elaborated on the dependencies between ontologies and distinguishes dependencies to *super-sub* and *referring-to* dependencies. Then it investigates the effects of an update operation at a dependency. Finally it discusses the management of ontologies from various perspectives (version control, updating and reusing).

3.6 On Preserving the Dependency Graph

The objective of the proposed model is the preservation of intelligibility of digital objects in a way compliant to the OAIS standard. So according to OAIS if we want to preserve an object (either digital or physical) for some particular Designated Community the main task we have to do is to package this object with additional Representation Information that allow interpreting the object. Now suppose that one adopts the approach proposed in this Chapter, i.e. objects

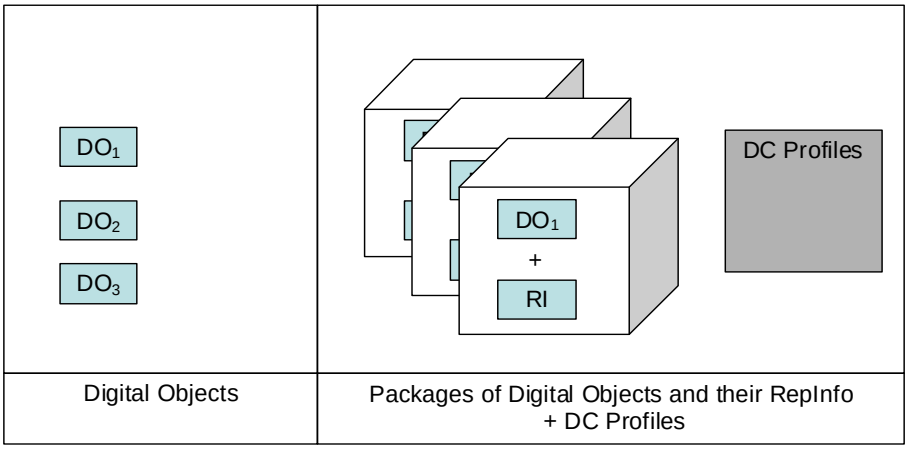

Figure 3.22: The contribution of OAIS for preserving the Intelligibility of Digital Objects

and their dependencies are recorded, the assumptions regarding the DC knowledge are expressed explicitly, and the contents of the package of an object (regarding Representation Information) are defined on the basis of object's dependencies and DC knowledge (i.e. based on intelligibility gap).

Suppose $\mathcal{T}$ is the set of the original digital objects that we want to preserve for a Designated Community C. By adopting our approach we result to a set PA of packages (one for each element of $\mathcal{T}$). In this case, there is no need to preserve the dependency graph that was used for deriving the contents of the packages. The only assumption is that C knows the OAIS approach and the packaging structure adopted. We cannot escape from this assumption. If on the other hand we want to be able to "adapt" the packages in PA for a different community C' then we have to preserve the dependency graph since this is required for producing $Gap(t, C')$ for each t in $\mathcal{T}$.

Let's now discuss the case where we want to preserve the dependency graph. We could follow a similar approach. In this case the dependency graph could be modeled as a module and the next step would be to identify what representation information (and how much) would be necessary for its preservation. This depends on the Designated Community. If the Designated Community is aware of the model (i.e. of the OAIS approach and the approach presented in this thesis) then there is no need to add any additional information since the dependency graph is already intelligible from that community. If on the other hand the community is unaware of these concepts, then Representation Information would be an object that documents sufficiently

these concepts (e.g. we could consider the PDF version of i.e. this thesis).

3.7 Summary

In this chapter we described a model for the preservation of intelligibility of digital objects. We described what are digital objects, what dependencies exist between them and how these dependencies affect the intelligibility of digital objects. We also discussed some OAIS-related concepts for the preservation of digital objects and how they are related with dependency management.

To this end we proposed the creation of a model based on the notions of module and dependency. The purpose of the model is the creation of a dependency graph for digital objects. Dependencies are used to represent which modules are required for the performability of various tasks. Additionally we proposed the modeling of the knowledge that is assumed to be known from designated communities, by associating to the users of those communities, the modules from the dependency graph that are known to them. Subsequently we defined a set of basic intelligibility-related services.

Besides we clarified the different interpretations that dependencies may have (conjunctive or disjunctive) and we described modeling and implementation frameworks for every case. Finally we described the design and implementation of `GapMgr` and we evaluated the applicability of the tool using experimental and real data.

We claim that the model is applicable to several domains, since it uses the very general notions of module and dependency. Furthermore the model is extensible and one can define specific module and dependency types according to his needs. This is really important since there is a plethora of different module types that exist and the set of tasks that can be performed on these modules cannot be fixed in advance. Regarding intelligibility-related services they can be exploited for: (a) enabling a disciplined method for deciding what metadata to include in a package, since it allows deriving packages that are intelligible for a certain community and the assumed knowledge of that community is explicitly specified, and (b) supporting the curation of existing packages, since the services could aid protecting archives from obsolescence risks and changes in the knowledge of the community.

Chapter 4

Provenance: Modeling and Querying

Provenance is the origin or the source and the history of the ownership of an object. Provenance information is well understood in the context of art where it refers to the documented history of an art object. Provenance can be of great importance in the context of digital world. The provenance information of digital objects refers to the documentation of the processes that led to the creation of a digital object. Moreover provenance information provides a great documentation for scientific products since it allows scientists to understand the product, ensure its validity and also allow its reproducibility.

Therefore provenance information for digital objects has to be properly recorded and archived. To this end we need conceptual models able to capture provenance information. The availability of such models can enable the exchange and integration of provenance data and can guide the design of provenance services. This chapter discusses the problem of modeling provenance and how it relates to the conceptualization of CIDOC CRM ontology. To this end an extension of CIDOC CRM is presented that is able to capture the modeling and query requirements regarding the provenance of digital objects.

4.1 Introduction to Provenance

According to Oxford English Dictionary[1] provenance is defined as *(i) the fact of coming from some particular source or quarter; origin, derivation. (ii) the history or pedigree of a work of art, manuscript, rare book, etc.; concretely, a record of the ultimate derivation and passage of an item through its various owners.* We want to further clarify the concept of provenance by

[1] http://www.oed.com/

interpreting it in an event-based manner. So provenance can be described from the events that occurred during the existence of a object. Examples of such events are the creation of an objet, the alteration of an object, the change of custody of an object etc.

The provenance of works of fine art, antiques and antiquities often assumes great importance. Documented evidence of provenance for an object can help to establish that it has not been altered and is not a forgery or a reproduction. Knowledge of provenance can help to assign the work to a known artist and a documented history can be of use in helping to prove ownership. The quality of provenance of an important work of art can make a considerable difference to its selling price in the market; this is affected by the degree of certainty of the provenance, the status of past owners as collectors, and in many cases by the strength of evidence that an object has not been illegally excavated or exported from another country. The provenance of a work of art may be recorded in various forms depending on context or the amount that is known, from a single name to an entry in a full scholarly catalogue several thousand words long.

Digital objects on the other hand change more frequently, are unlimitedly copied, are used as derivatives to produce new objects and are altered from several users. Additionally their creation and management requires complex processes that are usually not documented. For example Figure 4.1 shows an image describing the ozone concentration in the atmosphere derived from the ESA ERS-2 satellite. Some indicative questions regarding the provenance of this data product are: How this image has been derived? Who created this image and when? How it was created, with what processing algorithms, sensors etc?

Especially scientific research is generally held to be of good provenance when it is documented in detail. As vast amounts of scientific data are produced daily, their management is of prominent importance for e-science. Their provenance is more than necessary for documenting scientific results since it allows scientists to: (a) interpret, reproduce and validate a resulted product, (b) understand the derivation process, (c) identify which were the sources that led to the derivation of the product and (d) track who were the responsible users. So provenance captures of plethora of information about digital objects. In fact provenance may be as important (or even more) as the results.

In the context of scientific workflows[9, 16] provenance is a record of the derivation of a set of results. There are two distinct forms of provenance: prospective and retrospective. Prospective provenance captures a computational task's specification and corresponds to the steps that must be followed to generate a data product or class of data products. Retrospective provenance

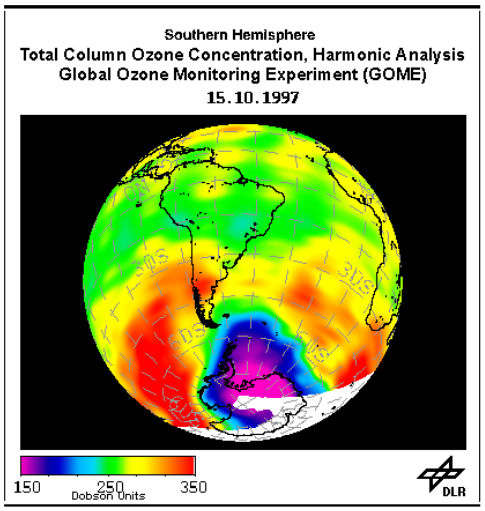

Figure 4.1: Global Ozone Monitoring Experiment (GOME) Image

captures the steps executed as well as information about the environment used to derive a specific data product. In other words prospective provenance is a recipe for the derivation of a data product while retrospective provenance is a detailed log of a computational task's execution. Also an important component of provenance is information about causality, which is the dependency relationships among data products and the processes that generate them. Causality can be inferred from both prospective and retrospective provenance and captures the sequence of steps which together with input data and parameters caused the creation of a data product.

4.1.1 Provenance and OAIS

The OAIS model[20] understands Provenance Information as a specific type of Context Information that documents the history of the Content Information. i.e. it tells the origin or source of the Content Information, any changes that may have taken place since it was originated, and who has had custody of it since it was originated. Examples of provenance information are the principal investigator who recorded the data, and the information concerning its storage, handling, and migration.

The middle column of Table 4.1 shows examples of OAIS Provenance (as listed in the standard) for various types of content information. The right column comments each row with respect to CIDOC CRM, while a more detailed discussion is given in the subsequent section.

Table 4.1: OAIS and CIDOC CRM Provenance

Content Information Type	OAIS Provenance	CIDOC CRM Provenance
Space Science Data	Instrument description	Context of observation/experiment
	Processing history	By whom, Derivation chain
	Sensor description	Context of observation/experiment
	Instrument	Context of observation/experiment
	Instrument mode	Context of observation/experiment
	Decommutation map	Context of observation/experiment
	Software interface specification	Context of observation/experiment
Digital Library Collections	For scanned collections:	For scanned collections:
	Metadata about the digitisation process	Context of digitisation process
	Pointer to master version	Derivation chain
	For born-digital publications:	For born-digital publications:
	Pointer to the digital original	Derivation chain
	Metadata about the preservation process	Context of preservation process
	Change history	Derivation chain
	Pointers to earlier versions of the collection item	Derivation chain
Software Package	Revision history	Derivation Chain
	License holder	By whom
	Registration	By whom
	Copyright	By whom

However OAIS does not propose any particular conceptual model or ontology. Moreover, it suggests that Preservation Description Information (PDI) should contain provenance information documenting the history of the data object (as illustrated in Figure 4.2). Again provenance, is just a box and no conceptual model is specified.

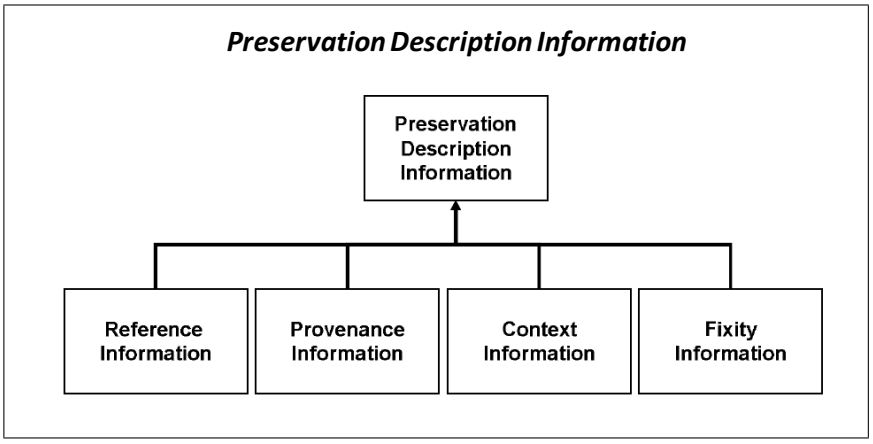

Figure 4.2: OAIS PDI Preservation Description Information

4.2 CIDOC CRM Extension for Digital Objects

CIDOC Conceptual Reference Model (ISO 21127) [21] is a core ontology of 80 classes and 132 relations describing the underlying semantics of over a hundred database schemata and structures from all museum disciplines, archives and libraries. It provides definitions and a formal structure for describing the implicit and explicit concepts and relationships used in cultural heritage documentation. CIDOC CRM is intended to promote a shared understanding of cultural heritage information by providing a common and extensible semantic framework that any cultural heritage information can be mapped to. CIDOC CRM is the result of long-term interdisciplinary work and agreement. It has been derived by integrating (in a bottom-up manner) hundreds of metadata schemas and is stable (almost no change the last 10 years). We could say that the basic design principles are (a) *empirical bottom-up knowledge engineering* and (b) *object-oriented modeling*. As regards the latter, CIDOC CRM has a rich structure of "intermediate" classes and relations, which apart from being very useful for building query services (enabling queries at various levels of abstraction and granularity), it makes its extension to other domains easier and reduces the risk of over-generalization/specialization. In essence, it is a generic model for recording the "what has happened" in human scale. It can generate huge, meaningful networks of knowledge by a simple abstraction: history as meetings of people, things and information. Figure 4.3 depicts the main concepts of CIDOC CRM.

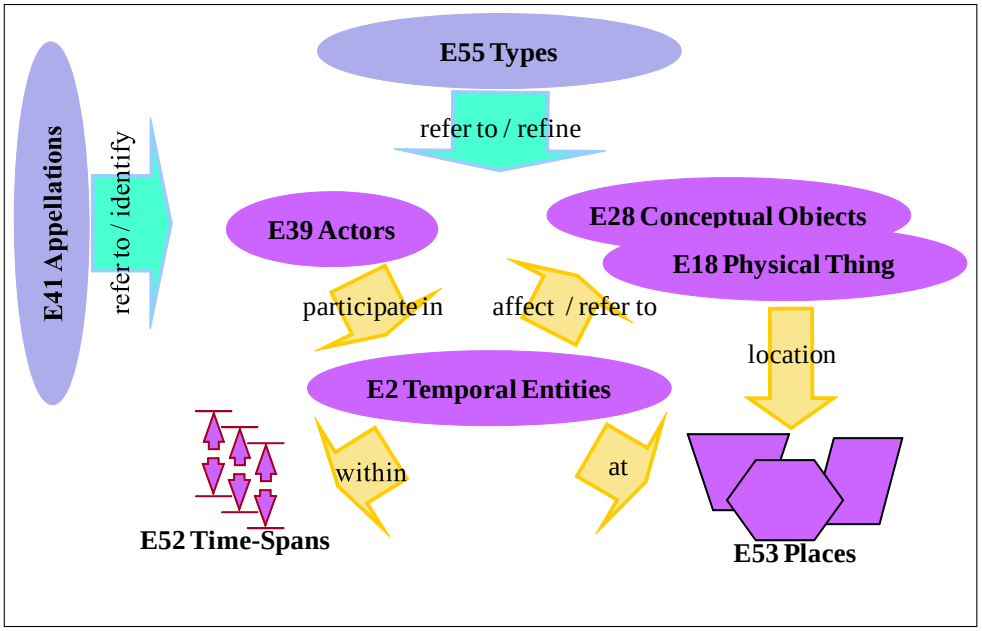

Figure 4.3: The main concepts of CIDOC CRM

Regarding the *modeling methodology*, we have taken as empirical evidence existing data structures from different domains, and analyzed the data structure elements for their underlying common conceptualization necessary to answer questions about the dependency of scientific data on tools, methods and relevant environmental factors of their creation, so that the data quality can be assessed and primary and secondary data can be reused or reprocessed for scientific purposes. The empirical evidence comes from scientific imaging for various purposes, satellite data, medical laboratory tests, physics experiments. In some scientific laboratories, there is not yet an established good practice with respect to complete provenance metadata, or the metadata are highly specific to a particular device. In these cases, our model allows for generalizing and complementing existing metadata creation practices. Top-down approaches, such as OPM [32], suffer from overgeneralization. For instance, due to neglecting the difference between material and immaterial items, OPM cannot describe errors introduced by failures of individual devices, such as dust on a sensor or partial data loss on a DVD.

Regarding the application of CIDOC CRM for scientific data, the idea is that scientific data and metadata can be considered as historical records. Scientific observation and machine-supported processing is initiated on behalf of and controlled by human activity. Things, data, people, times and places are causally related by events. Other relations are either deductions from events or found by observation. In brief, the basic *properties* that we wish to support regarding the extension and application of CIDOC CRM on digital objects are:

- Full *interpretability* of scientific or cultural data with respect to their meaning and quality, in particular all intended and unintended factors possibly influencing the outcome (environmental and hardware effects).

- Ability to *reprocess* primary or half-processed data with different parameters or different algorithms, in particular re-calibration.

- Ability to *trace all dependencies* for digital preservation, such as imminent obsolescence of software to display, process or migrate data. In addition, ability to *clean* reproducible intermediate results, to infer from processing steps *features preserved* between input and output, such as the motif of a digital image under a contrast readjustment ("get all images of this building").

- Ability to search for comparable data sets for integrated evaluation, such as for climate change studies.

Regarding *provenance-related query services*, in the context of CIDOC CRM they can be considered as queries that can take as input an object and answer questions of the form:

- **Context**
  - **by whom (either creator or responsible for creation)**
  - **of observation/experiment**
  - **of digitization**
- **Derivation chain**

The current version of CIDOC CRM (version 5.0.1) can support queries regarding the creator or the responsible for creation of an object ("by whom" type of queries) and examples are provided later on. However, the other two types of provenance queries ("context" and "derivation chain" queries) are not directly supported by the current version. For this reason below we describe an extension of CIDOC CRM ontology for capturing such cases. We will refer to this extension with the name $\texttt{CRM}_{dig}$.

4.2.1 Overview of the Extension

Figure 4.4 depicts an overview of the extensions as visualized by StarLion [42]. The diagram shows the new classes (in *blue*) and the directly referred objects from CIDOC CRM (in *yellow*). CIDOC CRM and $\texttt{CRM}_{dig}$ adopt the following naming conventions:

- **E**XX_Name denote Entities of CIDOC CRM
- **P**XX_Name denote Properties of CIDOC CRM
- **C**XX_Name denote Entities of $\texttt{CRM}_{dig}$
- **S**XX_Name denote Properties of $\texttt{CRM}_{dig}$

We have to note that the notion of a digital machine event, digital measurement and formal derivation are very generic, and the essence of e-science. The notion of digitization is specific to certain processes, and assists reasoning about "depicted objects". Similar specializations may be created to reason about other measurement devices.

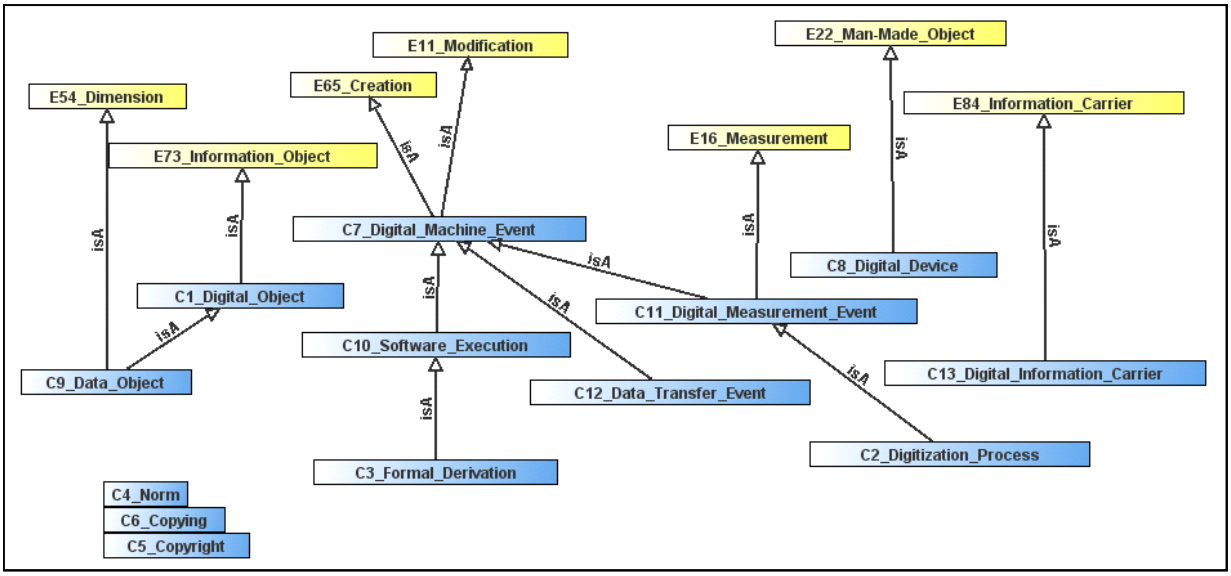

Figure 4.4: CIDOC CRM Digital ($\texttt{CRM}_{dig}$)

4.2.2 Detailed Description of the New Classes

Below we give a detailed description of the classes of $\texttt{CRM}_{dig}$. We have defined four new specializations of material and immaterial items and six new specializations of events:

- **C1 Digital Object**, which comprises identifiable immaterial items, that can be represented as sets of bit sequences, such as data sets, e-texts, images, audio or video items, software, etc., and are documented as single units. Any aggregation of instances of **C1 Digital Object** into a whole treated as single unit is also regarded as an instance of **C1 Digital Object**. This means that for instance, the content of a DVD, an XML file on it, and an element of this file, are regarded as distinct instances of C1 Digital Object, mutually related by the ***P106 is composed of (forms part of)*** CIDOC CRM property. A **C1 Digital Object** does not depend on a specific physical carrier, and it can exist on one or more carriers simultaneously.

- **C2 Digitization Process**, which comprises events that result in the creation of instances of **C9 Data Object** that represent the appearance and/or form of an instance of **E18 Physical Thing** such as paper documents, statues, buildings, paintings, etc. A particular case is the analogue-to-digital conversion of audiovisual material. This class represents the transition from a material thing to an immaterial representation of it. The characteristic subsequent processing steps on digital objects are regarded as instances of **C3 Formal**

Derivation.

- **C3 Formal Derivation**, which comprises events that result in the creation of a **C1 Digital Object** from another one following a deterministic algorithm, such that the resulting instance of digital object shares representative properties with the original object. In other words, this class describes the transition from an immaterial object referred to by property ***S21 used as derivation source (was derivation source for)*** to another immaterial object referred to by property ***S22 created derivative (was derivative created by)*** preserving the representation of some things but in a different form. Characteristic examples are colour corrections, contrast changes and resizing of images.

- **C7 Digital Machine Event**, which comprises events that happen on physical digital devices following a human activity that intentionally caused its immediate or delayed initiation and results in the creation of a new instance of **C1 Digital Object** on behalf of the human actor. The input of a **C7 Digital Machine Event** may be parameter settings and/or data to be processed. Some **C7 Digital Machine Events** may form part of a wider **E65 Creation** event. In this case, all machine output of the partial events is regarded as creation of the overall activity.

- **C8 Digital Device**, which comprises identifiable material items such as computers, scanners, cameras, etc. that have the capability to process or produce instances of **C1 Digital Object**.

- **C9 Data Object**, which comprises instances of **C1 Digital Object** that are the direct result of a digital measurement or a formal derivative of it, containing quantitative properties of some physical things or other constellations of matter.

- **C10 Software Execution**, which comprises events by which a digital device runs a software program or a series of computing operations on a digital object as a single task, which is completely determined by its digital input, the software and the generic properties of the device.

- **C11 Digital Measurement Event**, which comprises actions measuring physical properties using a digital device, that are determined by a systematic procedure and creates an instance of **C9 Data Object**, which is stored on an instance of **C13 Digital Information Carrier**. In contrast to instances of **C10 Software Execution**, environmental

factors have an intended influence on the outcome of an instance of **C11 Digital Measurement Event**. Measurement devices may include running distinct software, such as the RAW to JPEG conversion in digital cameras. In this case, the event is regarded as instance of both classes, **C10 Software Execution** and **C11 Digital Measurement Event**.

- **C12 Data Transfer Event**, which comprises events that transfer a digital object from one digital carrier to another. Normally, the digital object remains the same. If in general or by observation the transfer implies or has implied some data corruption, the change of the digital objects may be documented distinguishing input and output rather than instantiating the property ***S14 transferred (was transferred by)***.

- **C13 Digital Information Carrier**, which comprises all instances of **E84 Information Carrier** that are explicitly designed to be used as persistent digital physical carriers of instances of **C1 Digital Object**. A **C13 Digital Information Carrier** may or may not contain information, e.g., an empty diskette.

4.2.3 Indicative Examples

4.2.3.1 The Provenance of GOME dataset

To introduce the basic concepts of $\texttt{CRM}_{dig}$, we adopt a real world scenario coming from ESA (European Space Agency). The GOME (Global Ozone Monitoring Experiment) dataset, consists of data captured from a sensor on board the ESA ERS-2 (European Remote Sensing) satellite. In general the Satellite (having various properties like name, id) is placed in a particular Orbit (e.g. geosynchronous) and is equipped with a number of Sensors. The captured measurements are sent to a ground earth acquisition station (e.g. at the Kiruna Station), transfered to an Archiving Facility (at ESA-ESRIN) for long term preservation and to a Processing Facility (at DLR - German Aerospace Center) for various data transformations that yield various kinds of Products. Data sets are distinguished according to their processing level to: Level 0 (raw data), Level 1 (radiances/reflectances), Level 2 (geophysical data as trace gas amounts), and Level 3 (a mosaic composed by several level 2 data with interpolation of data values to fill the satellite gaps). Figure 4.5 illustrates the trail of the GOME data.

Below we describe how this scenario is modeled according to $\texttt{CRM}_{dig}$. Figure 4.6 shows how data capturing, transmission, processing and archiving events are modelled with $\texttt{CRM}_{dig}$, together

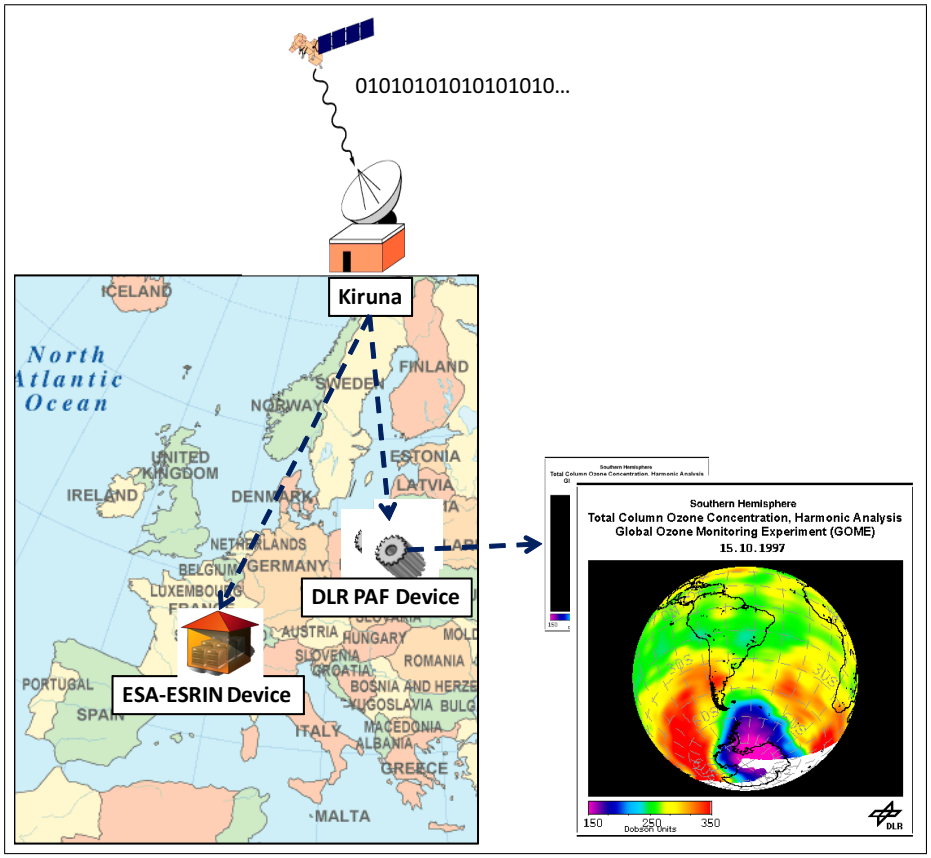

Figure 4.5: The trail of GOME data scenario

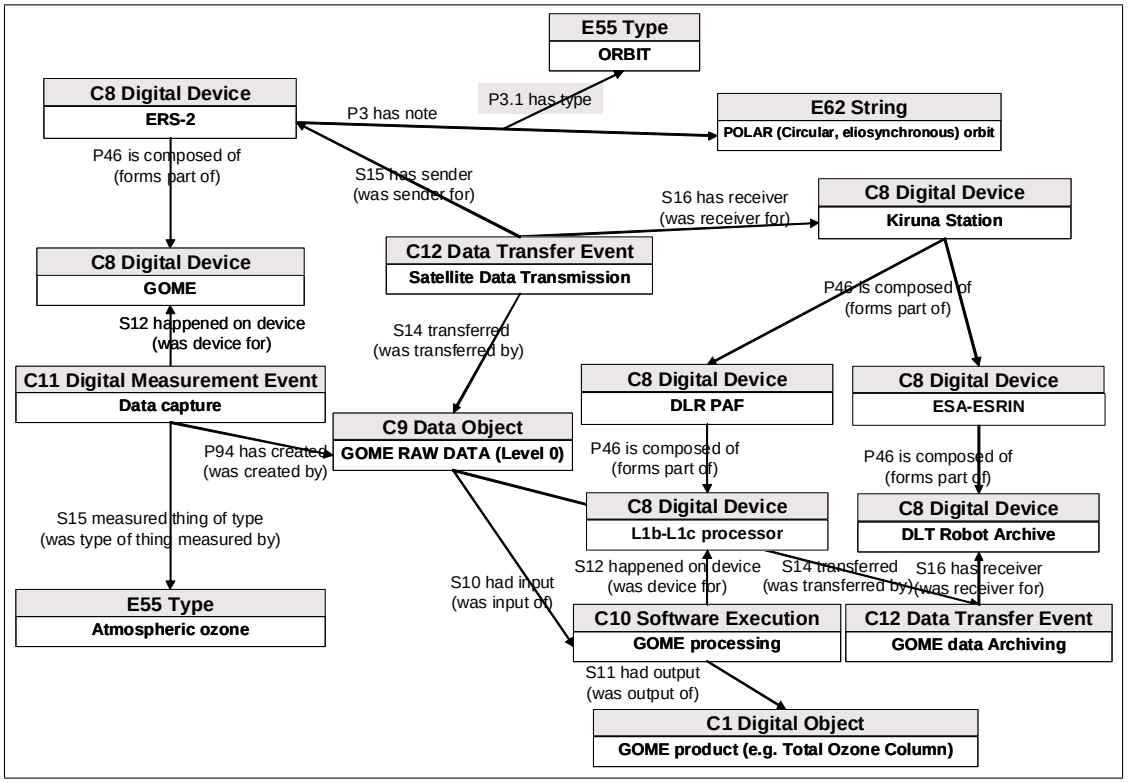

Figure 4.6: The trail of GOME data scenario modeled with $\texttt{CRM}_{dig}$

with their related products. We adopt a graphical language similar to UML Object Diagrams. The ESA ERS-2 Satellite is modelled as a **C8 Digital Device** whose orbit is recorded in a **E62 String** of **E55 Type** "ORBIT" through the *P3 has note* property and is related through the *P46 is composed of (forms part of)* relationship with Sensors which are also **C8 Digital Device** instances. The data capturing event is modelled as a **C11 Digital Measurement Event** which relates to a Sensor through the *S12 happened on device (was device for)* property and records what it measures through the *S15 measured thing of type (was type of thing measured by)* property. The result of the data capturing event is the creation of a GOME RAW DATA (Level 0) data set, modelled as a **C9 Data Object** and linked to the data capturing event through the *P94 has created (was created by)* property.

The ESA ERS-2 Satellite data transmission to the Kiruna Station is modelled as a **C12 Data Transfer Event**. The data transmission relates to the ESA ERS-2 Satellite through the *S15 has sender (was sender for)* property, to the GOME RAW DATA (Level 0) data set through the *S14 transferred (was transferred by)* property and to the Kiruna Station through the *S16 has receiver (was receiver for)* property. The Kiruna Station is modelled as a **C8 Digital Device** which is *P46 composed of (forms part of)* two devices the DLR PAF Device and ESA-ESRIN Device which are also modeled as **C8 Digital Devices**. The DLR PAF is *P46 composed of (forms part of)* the L1b-L1c processor (**C8 Digital Device**) and the ESA -ESRIN is *P46 composed of (forms part of)* the DLT Robot Archive (**C8 Digital Device**) respectively.

GOME processing is modelled as a **C10 Software Execution** which receives as input, through the *S10 had input (was input of)* property, the GOME RAW DATA (Level 0) data set and produces the L0 GOME product (e.g. Total Ozone Column) **C1 Digital Object** (*S11 had output (was output of) property*). GOME data archiving is an event modelled as a **C12 Data Transfer Event** that relates to the DLT Robot Archive through the *S16 has receiver (was receiver for)* property and to the GOME RAW DATA (Level 0) data set through the *S14 transferred (was transferred by)* property.

Figure 4.7 shows how the transformation of $L0 \rightarrow L1 \rightarrow L2$ products can be modeled using $\texttt{CRM}_{dig}$.

4.2.3.2 Conversion

Here we describe how we can model activities that result in the creation of a digital object from another one, following a deterministic algorithm, as Formal Derivations. Formal Derivation

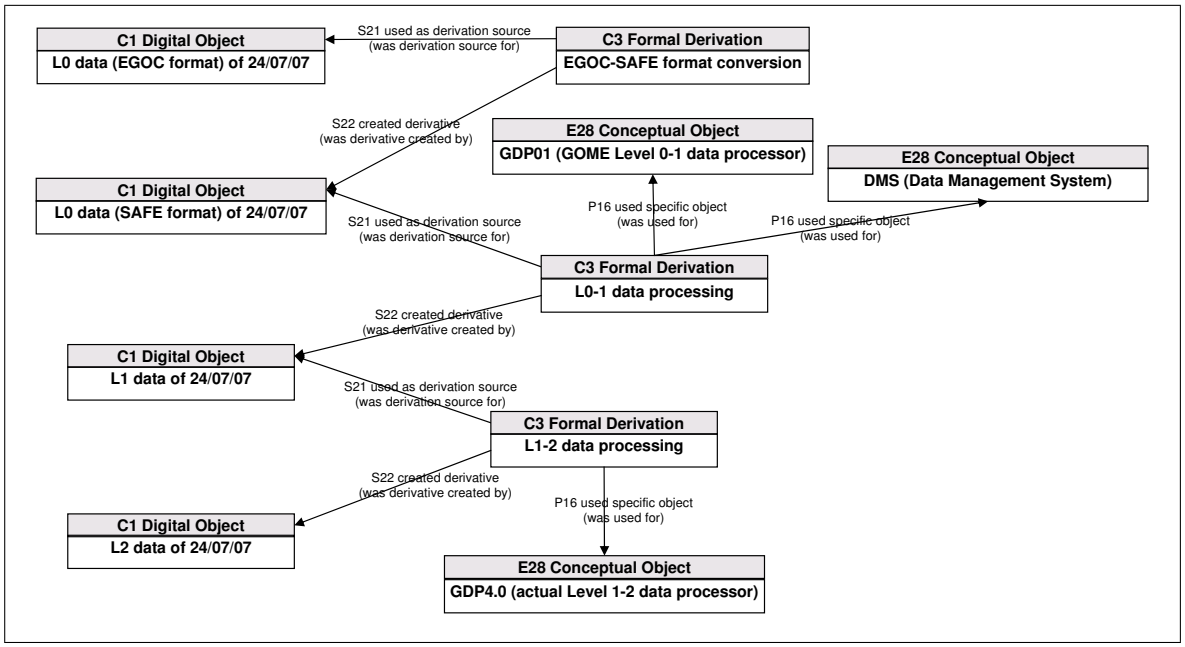

Figure 4.7: Modeling the data processing levels of GOME

represents the transition from an immaterial object to another immaterial object. The resulting instance of digital object shares representative properties with the original object and can be mechanically reproduced.

For instance, suppose we have a converter called **JPG2PNG** and consider three photographs **Crete.jpg**, **Crete.png** and **CreteSmall.png**. The latter two derived from the first photograph by using the converter. **CreteSmall.png** has lower resolution. Figure 4.8 and Figure 4.9 illustrates how the above scenario can be modeled using $\texttt{CRM}_{dig}$. In the first case the converter is used to produce **Crete.png** from **Crete.jpg** and then Adobe Photoshop application is used to reduce the resolution of **Crete.png** and produce **CreteSmall.png**. In the second case both **Crete.png** and **CreteSmall.png** are produced from **Crete.jpg** by using the converter with different parameters.

In both cases, the photographs are instances of **C1 Digital Object**. The class **E55 Type** is used to denote the format of each photograph (see Figure 4.8) while classes **E54 Dimension** and **E60 Number** are used to model the resolution of each photograph. In general these classes may be used in order to model digital object parameters and their respective values.

The conversion event is an instance of **C3 Formal Derivation** which through the link ***P33_used_specific_technique (was used by)*** points to the specific algorithm used for the conversion (instance of class **E29 Design or Procedure**). In this example the parameters with which the converter was called are not modeled as separate entities but are implied in the name

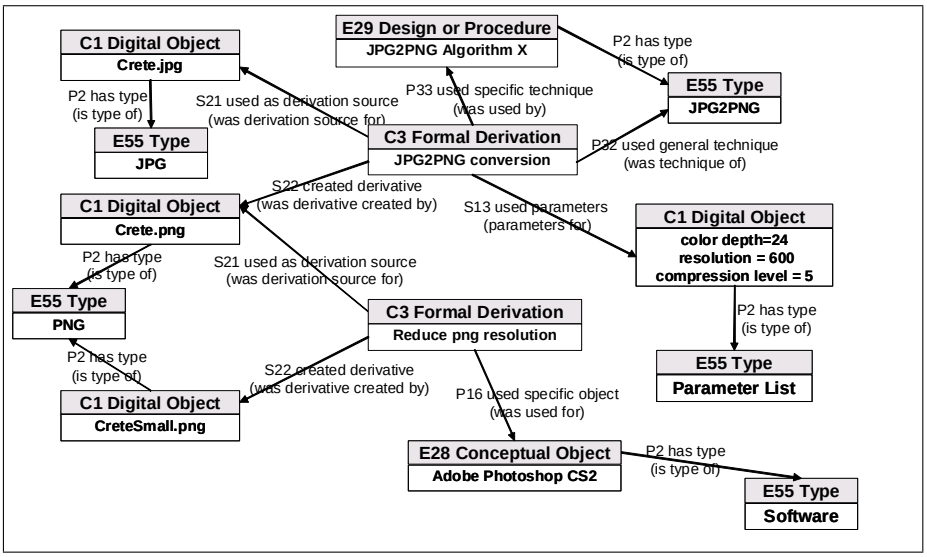

Figure 4.8: JPG2PNG Converter

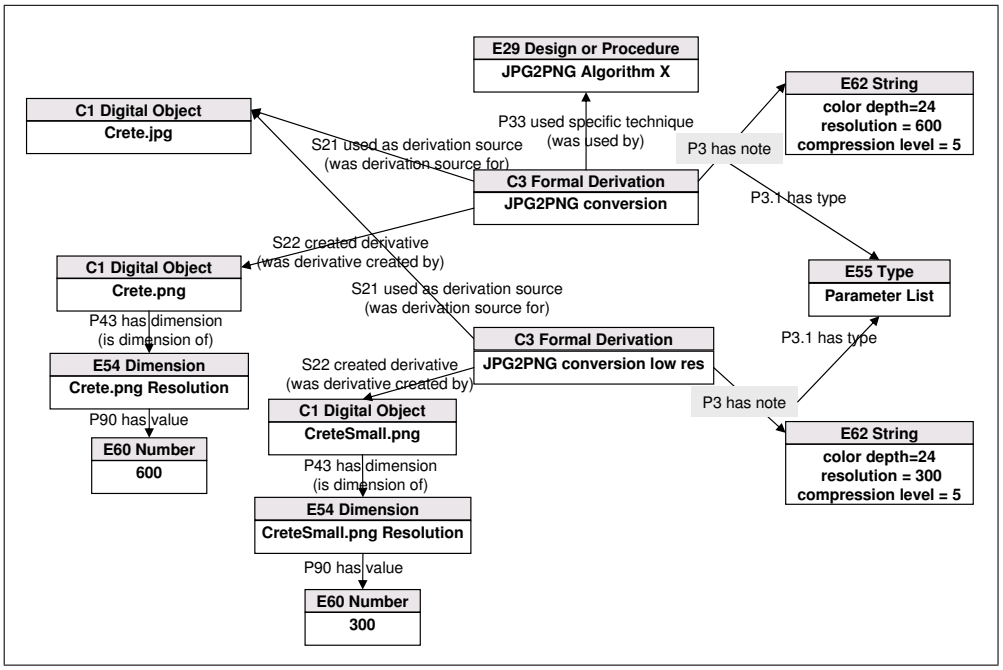

Figure 4.9: JPG2PNG Converter

of the **E29 Design or Procedure** instance. Since **E29 Design or Procedure** can be linked with other **E29 Design or Procedure** through ***P69 is associated with***, we can associate the specific call of a converter with the generic converter. **C3 Formal Derivation** is linked to **E55 Type** through ***P32 used general technique (was technique of)*** and denotes the generic type of the conversion. In Figure 4.9 we can see how the different parameter list is modeled through the use of property ***P3 has note*** that points to an instance of class **E62 String**.

4.2.3.3 Emulation

Another example of formal Derivation is *emulation*. Figure 4.10 shows an example of modeling the Handy emulator. To play Atari Lynx games on a Windows-based PC by using the Handy emulator the file `LYNXBOOT.IMG` is needed as well as Lynx game ROMs. A specific instance of the Handy emulator is an instance of class **E29 Design or Procedure**. The emulation activity is an instance of **C3 Formal Derivation** which has the property ***S21 used as derivation source*** pointing to the file `LYNXBOOT.IMG` which is an instance of **C1 Digital Object**. The result of the emulation is an instance of **E29 Design or Procedure**. Issues regarding the principles of designing emulators or the reasoning on the properties of emulations, e.g. the Universal Virtual Computer (UVC) [27, 45], go beyond the scope of our work.

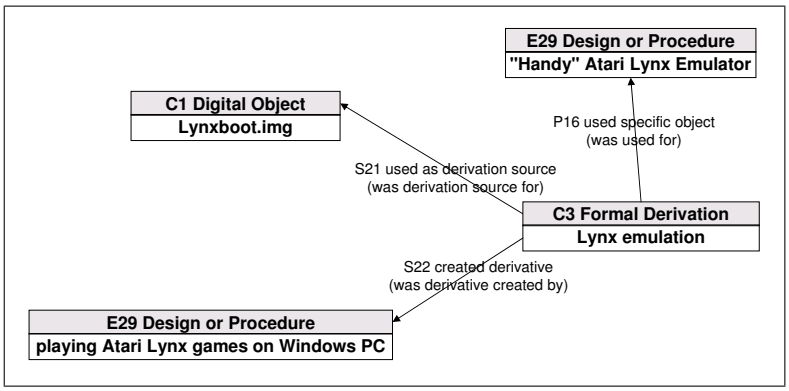

Figure 4.10: Modeling Emulation

4.3 Provenance Queries over $\texttt{CRM}_{dig}$

Queries regarding provenance, could be based on paths of $\texttt{CRM}_{dig}$. We can identify the following query requirements:

- Get the creator of an object

- Get the earlier versions of an item
- Get the events that changed the custody of an item
- Get the master version of an object
- Get the scanner/resolution of a digital object

Table 4.2 provides an indicative list of such queries. They can be considered as general purpose query templates that can be refined according to needs. Each template has a name, it takes as input a type-restricted resource (e.g. an instance of **E28 Conceptual Object**), and returns as output a number of typed resources (of course, as in any object oriented system, the type of the actual input/output parameters can be a subtype of the one specified in the template). For each template the path over the semantic graph that should be followed for computing the answer is specified in the form of a sequence consisting of consecutive "edges" of the form:

$$SourceClassName \rightarrow PropertyName \rightarrow TargetClassName$$

Some of these templates are recursive. For instance, consider template number 5:

{

$E29\_Design\_or\_Procedure \rightarrow P94B\_was\_created\_by \rightarrow E65\_Creation \rightarrow$

$P15F\_was\_influenced\_by \rightarrow E29\_Design\_or\_Procedure$

}$*$ repeat until $P15F\_was\_influenced\_by$ is null

This query comprises an expression that takes as input an instance of E29 and returns another instance(s) of E29 (those influenced by) and this is continued recursively until there is no other P15F property that could be followed.

Figures 4.11 and 4.12 present the $\texttt{CRM}_{dig}$ graphs for the query templates 1 and 6 respectively.

4.4 Related Work on Modeling Provenance

A related model for provenance is Open Provenance Model (OPM) [32]. In brief OPM allows characterizing what caused things to be i.e. how things depended on others and resulted in a specific state. The ontology that is defined by OPM is minimal and it consists of 3 classes: *Artifact*, which is an immutable piece of state, *Process*, describing the actions that resulted to the existence of an artifact and *Agent*, that are contextual entities acting as a catalyst of

Table 4.2: Provenance Query templates over $\texttt{CRM}_{dig}$

#	Description	Input	Output	Path
1	Get the Creator of a Digital Object	A Digital Object Instance of E28_Conceptual_Object	Instance of E82_Actor_Appellation	$E28\_Conceptual\_Object \rightarrow P94B\_was\_created\_by \rightarrow E65\_Creation \rightarrow P14F\_carried\_out\_by(P14.1\_in\_the\_role\_of \rightarrow E55\_Type = Developer) \rightarrow E39\_Actor \rightarrow P131F\_is\_identified\_by \rightarrow E82\_Actor\_Appellation$
2	Get the Scanner used to capture a Digital Image	A Digital Image Instance of C1_Digital_Object	Instance of C8_Digital_Device	$C1\_Digital\_Object \rightarrow S11B\_was\_output\_of \rightarrow C7\_Digital\_Machine\_Event \rightarrow S12F\_happened\_on\_device \rightarrow C8\_Digital\_Device$
3	Get the Resolution of a Digital Object	A Digital Object Instance of E73_Information_Object (Digital Image)	Instance of E60_Number	$E73\_Information\_Object \rightarrow P39B\_was\_measured\_by \rightarrow C2\_Digitization\_Process \rightarrow P40F\_observed\_dimension \rightarrow E54\_Dimension \rightarrow P90F\_has\_value \rightarrow E60\_Number$
4	Get the Master Version of a Digital Object	A Digital Object Instance of E73_Information_Object (Digital Image)	Instance of E18_Physical_Thing	$E73\_Information\_Object \rightarrow P94B\_was\_created\_by \rightarrow C2\_Digitization\_Process \rightarrow S1F\_digitized \rightarrow E18\_Physical\_Thing$
5	Get Earlier Versions of a Digital Derivative	A Digital Derivative Instance of E29_Design_or_Procedure	List of Instances of E29_Design_or_Procedure	$\{E29\_Design\_or\_Procedure \rightarrow P94B\_was\_created\_by \rightarrow E65\_Creation \rightarrow P15F\_was\_influenced\_by \rightarrow E29\_Design\_or\_Procedure\} *$ $repeat\ until\ P15F\_was\_influenced\_by\ is\ null$
6	Get the custody history of an Object	An Object Instance of E84_Information_Carrier	List of Instances of E82_Actor_Appellation	$E84\_Information\_Carrier \rightarrow P50F\_has\_current\_keeper \rightarrow \{E39\_Actor \rightarrow P29B\_received\_custody\_through \rightarrow E10\_Transfer\_of\_Custody \rightarrow P28F\_custody\_surrendered\_by \rightarrow E39\_Actor\} *$ $repeat\ until\ P29B\_received\_custody\_through\ is\ null \rightarrow P131F\_is\_identified\_by \rightarrow E82\_Actor\_Appellation$

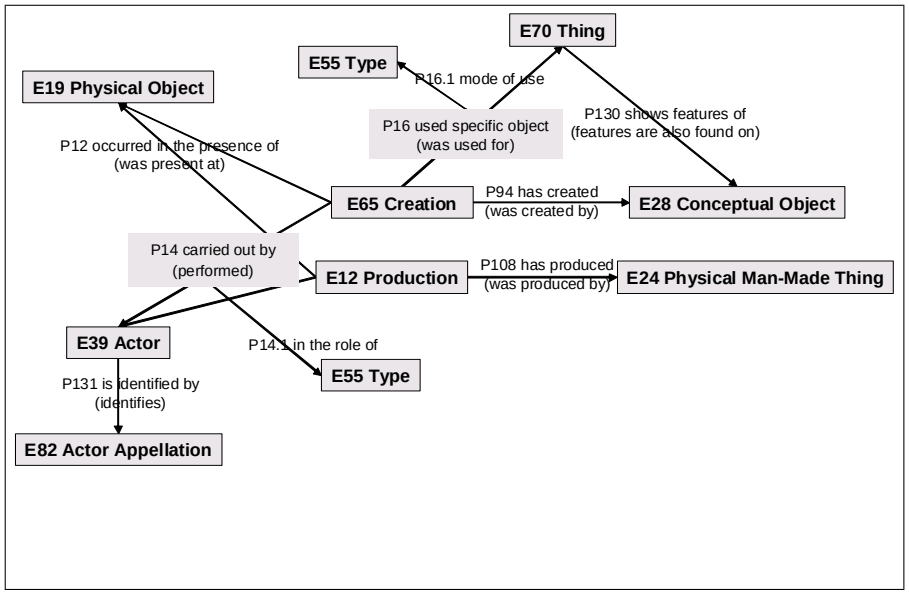

Figure 4.11: Sample query 1 - Find creator/producer

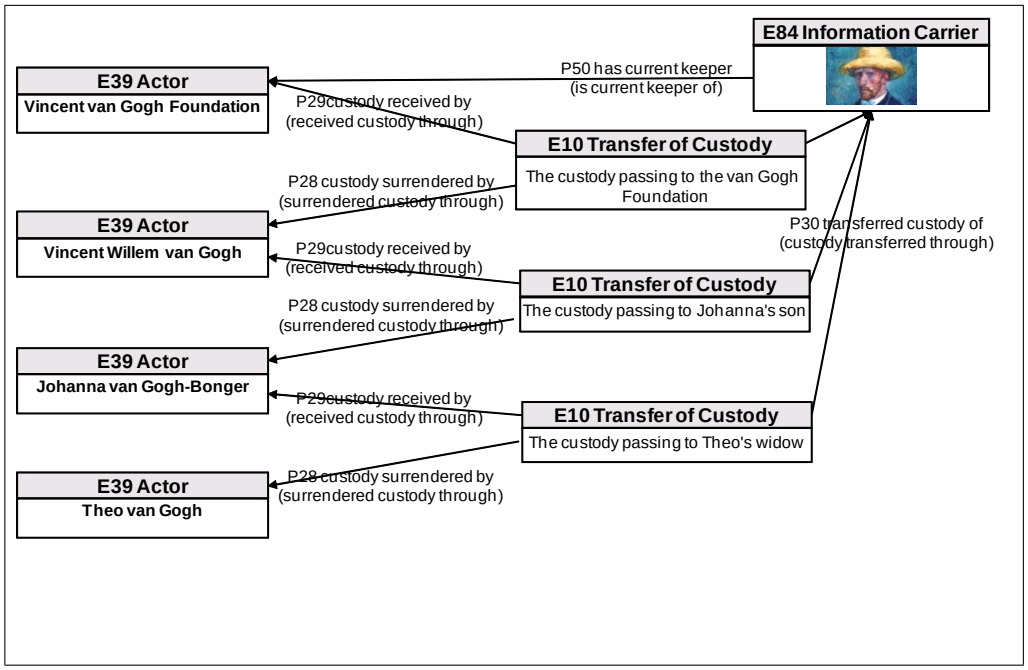

Figure 4.12: Sample query 6 - Change of custody chain

a process. Moreover 5 associations between these classes are defined *(used, wasGeneratedBy, wasControlledBy, wasTriggeredBy, wasDerivedFrom)*. OPM does not make any distinction between digital and physical objects and their provenance is represented by an annotated causality graph.

According to OPM, a process is considered as the set of actions performed on or caused by artifacts and resulting in new artifacts. Processes are connected with artifacts using *used* and *wasGeneratedBy* edges. By connecting a process to several artifacts by used edges, we are not just stating the individual inputs to the process. We are asserting that a causal dependency expressing that the process could take place and complete only because all these artifacts were available. Therefore we cannot model the participation of artifacts in the derivation history of an object without implying causality. On the other hand CIDOC CRM Digital contains a set of properties for modeling the participation of objects or persons in the derivation history of an object, without necessarily implying causality. For example for modeling the participation of a person in the derivation of an object without a specific role the property *P11 had participant (participated in)* can be used. If a person had an active participation (i.e. he was the responsible person for the digitization of an artifact) then the more concrete property *P14 carried out by (performed)* can be used. The same pattern occurs with the participation of objects where the properties *P12 occurred in the presence of (was present at)* and *P16 used specific object (was used*

for) can be used for modeling active and passive participation correspondingly. Therefore OPM does not capture non causal relationships. In order to capture such extra information, which is typically required for interoperability purposes, OPM uses *annotations*. An annotation instance will contain the annotable entity and a non-empty list of property-value pairs describing several properties of the entity. However the modeling of this extra information according to CIDOC CRM (using its rich structure of classes and properties), compared to OPM annotations, offers more expressibility and also enables the efficient querying of related information. Additionally the information that must be recorded might be related to several artifacts of an OPM graph. However annotations cannot be related, so the annotations of an OPM graph may contain redundant information.

It follows that, from the perspective of representation adequacy, provenance information recorded according to $\texttt{CRM}_{dig}$ can be mapped to an OPM-based view, but not the other way around. The main vision of OPM is to provide a vision of data flowing across systems. $\texttt{CRM}_{dig}$ allows modeling the derivation history of a digital object, as well as several information regarding the context of the derivation (i.e. digital devices that were used, responsible users, etc.). The class **C3 Formal Derivation** defined in $\texttt{CRM}_{dig}$ describes the transition of an immaterial object to another immaterial object. Therefore this class is adequate for capturing the data simulation that is derived from programs or workflows. In addition this class is also a subclass of class **E5 Event**. The ontology assumed by OPM does not explicitly model the concept of *Event*. Events are important since they enable the integration of information that concern an object (i.e. where an Event take place). For example the event of taking a photo comprehends various information about the conditions under which the photograph was taken, i.e. when, where, with what instruments, with what participants, for which reasons, etc., rather than focusing only on the process of capturing the image. Nevertheless, we should say that the way OPM treats processes resembles events (however the corresponding ontological structure of OPM is not rich).

In addition, OPM proposes a number of inference rules. Some of these are equivalent to the inferences due to the specialization relationships of CIDOC CRM extension. Some other could be expressed over the CIDOC CRM ontology by adopting an appropriate Rule Language. As an example, [31] describes an extension of the original CIDOC CRM for Interactive Multimedia Performances (IMP) enriched with temporal rules.

For example Figure 4.13 shows the derivation history of a digital object. Specifically a provenance graph according to OPM is shown. Ellipses are used to denote artifacts, rectangles

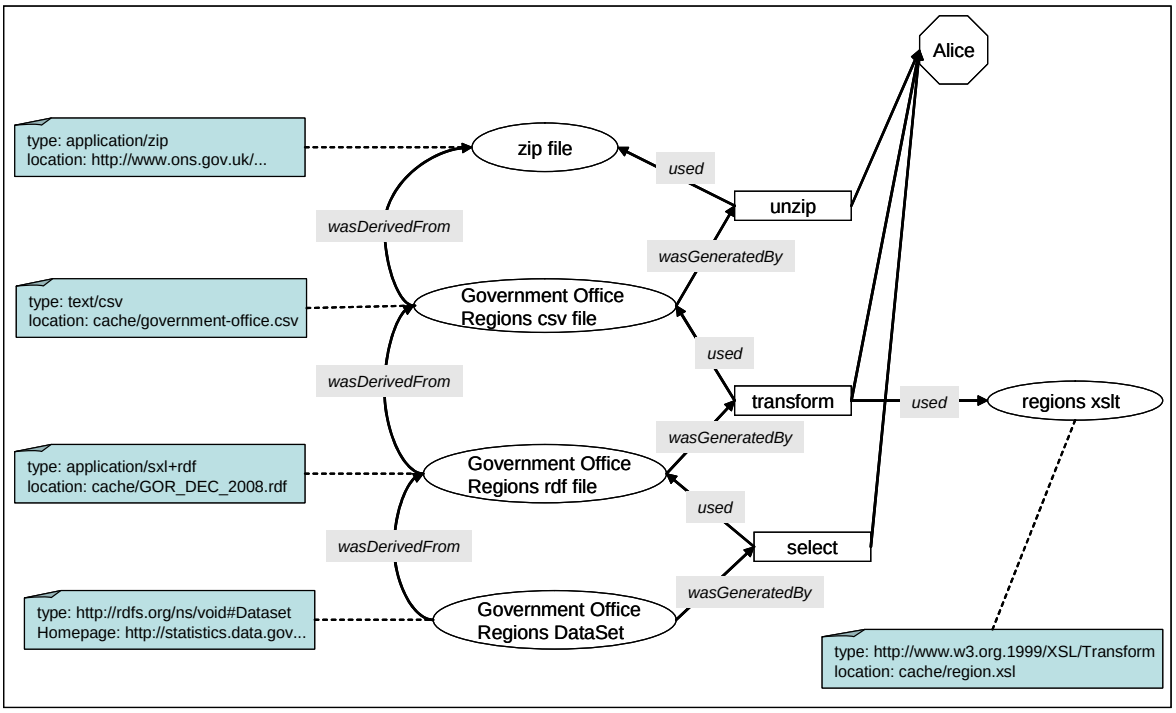

Figure 4.13: Provenance graph according to OPM

denote processes and the diamond for denoting the responsible users. Edges are used to denote the casual dependencies between OPM nodes. Any additional information is modeled using annotations (i.e. the location of the artifacts). Figure 4.14 shows how the OPM graph of Figure 4.13 can be modeled using $\texttt{CRM}_{dig}$. OPM processes are modeled as instances of `C3 Formal Derivation` and are connected with the used and derivative objects using the properties *S21 used as derivation source* and *S22 created derivative* respectively. Furthermore the information that is modeled using OPM annotations is modeled the appropriate classes of CIDOC CRM and $\texttt{CRM}_{dig}$.

4.5 Summary

In this section we described an extension of CIDOC CRM ontology, called $\texttt{CRM}_{dig}$, which is able to capture the modeling and query requirements regarding the provenance of digital objects. We also discussed the relationship with OAIS and described the representation of this ontology using RDF/S. Finally some indicative examples and query patterns were provided. The proposed model has higher general coverage and deeper specialization than OPM and is

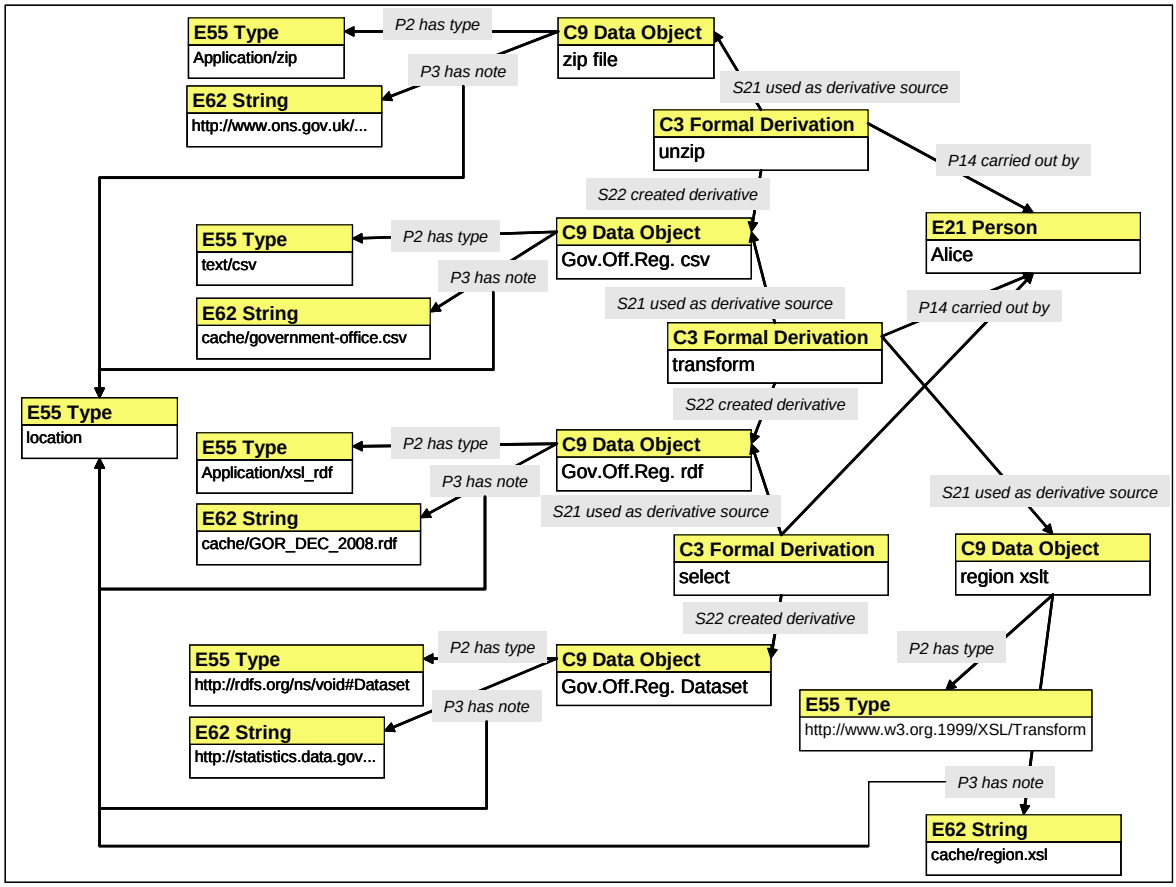

Figure 4.14: Provenance graph according to $\texttt{CRM}_{dig}$

applicable to all e-science domains.

Chapter 5

Automating the Ingestion and Transformation of Metadata

5.1 Introduction

The preservation of digital objects is a topic of prominent importance for archives and digital libraries. However the creation and maintenance of metadata is a laborious task that does not always pay off immediately. For this reason there is a need for tools that automate as much as possible the creation and curation of preservation metadata. Our objective is to bypass the strict (often manual) ingestion process while at the same being compatible with it. According to the traditional approach, the ingestion phase starts with assigning identifiers to the objects and then extracting/creating metadata for these objects. These metadata can be expressed using various metadata schemas and formats (like DIDL, METS, etc) and usually the update/movement of a metadata record is prohibited. We want to relax these constraints and automate the process of metadata extraction and management.

We propose an approach where some of the metadata are extracted automatically and periodic re-scans are used for keeping them up-to-date. The user is able to view the extracted metadata and add additional knowledge. PreservationScanner (`PreScan` for short) is a tool we have developed for the problem at hand. `PreScan` is quite similar in spirit with the crawlers of Web search engines. In our case we scan the file system, we extract the embedded metadata and build an index. In contrast to web search engine crawlers we want to: (a) support more advanced extraction services, (b) allow the manual enrichment of metadata, (c) use more expressive representation frameworks for keeping and exploiting the metadata (i.e. metadata

schemas expressed in Semantic Web languages), (d) offer rescanning services that do not start from scratch but exploit the previous status of the index, and (e) associate the extracted metadata with other sources of knowledge (i.e. registries of formats). The latter is important since the *intelligibility* of a digital object depends on the availability of other digital objects. To this end we propose linking objects with external sources and providing dependency management services for identifying obsolescence risks.

5.2 Metadata and Preservation Requirements

Metadata can be described as *"structured, encoded data that describe characteristics of information-bearing entities to aid in the identification, discovery, assessment, and management of the described entities"* [1]. Table 5.1 shows some examples of what can be considered as metadata.

Table 5.1: Examples of Metadata

Type	Metadata
Book	title, author, date of publication, subject, ISBN, dimensions, number of pages, text language
Photographs	date and time of capture, details of the camera settings (focal length, aperture, exposure), coordinates (geotagging)*
Audio files	album name, song title, genre, year, composer, contributing artist, track number and album art
Relational Databases	Database catalog storing information about the names of tables and columns, the domains of values in columns, number of rows, etc
Software	For example, in Java, the class file format contains metadata used by the Java compiler and the Java virtual machine to dynamically link classes and to support reflection.
Web Pages	Meta-tags, general purpose descriptions expressed using Semantic Web languages.

*: Many digital cameras record metadata in exchangeable image file format (EXIF)

Metadata can be stored either *internally*, i.e. in the same file as the data, or *externally*, i.e. in a separate file. The former are usually called *embedded*, the latter *detached*. The detached metadata are usually stored in a special repository. Both approaches have advantages and disadvantages. One benefit of the embedded metadata is that they are transferred with the data and thus their access and manipulation is straightforward. However embedded metadata can create redundancies and this approach does not allow holding and managing all metadata

[1] American Library Association, Task Force on Metadata Summary Report, June 1999

together. On the other hand, if the metadata are detached, then this means that they are stored in a special repository. This approach has less redundancies, we can support efficient metadata search, and we can manipulate them efficiently, e.g. we can perform bulk metadata updates. However, the way metadata are linked to data should be treated with care as inconsistencies may arise. Overall, to manage a corpus of digital objects requires tackling several issues and problems. From our experience and analysis, we have identified the following basic requirements:

- **Automatic Scanning** of file systems

  We need systems that can operate like the crawlers of Web search engines, i.e. scan the desired parts of the file system (or network) according to a given configuration (regarding desired folders, extensions of files, etc).

- **Automatic Format Identification** and **Extraction of Embedded Metadata**

  It is useful to extract the embedded metadata so that to make them visible to the curators. Since several formats contain embedded metadata we need to extract them easily. Various format identification tools and metadata extractors have been developed and can be used for this purpose (e.g. JHOVE[2], meta-extractor[3], Droid[4]).

- Support for **Human-entered/edited Metadata**

  Users (or curators) should be able to add extra metadata to an already scanned file (apart from the automatically extracted). For example one might want to add extra metadata about provenance, digital rights, or the context of the objects.

- **Periodic Re-Scannings** without loosing the human-provided metadata

  As the contents of the files change frequently we need to update their metadata in a flexible manner. To this purpose, periodic re-scannings are useful for ensuring the freshness of the metadata. However the human-provided metadata should be preserved.

- **Referential Integrity** services

  If a file/folder is moved to another location we would like to identify such changes in order to reflect them to its detached metadata. This is important also for ensuring that the human-entered metadata will be preserved.

[2]http://hul.harvard.edu/jhove
[3]http://meta-extractor.sourceforge.net
[4]http://droid.sourceforge.net/wiki/index.php

- **Dependency Management** and **Intelligibility-preservation** services

  A digital object might depend directly or indirectly on other modules. A module can be either a hardware/software component, a file format etc. Therefore we need to record such dependencies in order to facilitate tasks like: (a) the identification of the objects that are in danger in case a module (e.g. a software component or a format) is becoming obsolete (or has been vanished), (b) the decision of what metadata need to be captured and stored for a designated community, and (c) the reduction of the metadata that have to be archived, or delivered (as a response to queries) to the users of that community.

- Exploitation of **Existing Registries**

  External registries (like Pronom [39], GDFR [33], Registry of CASPAR) that contain information about file formats and versions of software for each one of them, should be exploited.

- **Usability and Control of the Whole Process**

  The functionalities must be performed in a simple and easy to use manner with the support of graphical user interfaces.

5.3 `PreScan` Tool

PreservationScanner, for short `PreScan`, is a system that we have developed based on the previously described requirements. `PreScan` consists of four main components: the **scanner**, a component responsible for scanning file systems, the **metadata extractor**, a component for extracting the embedded metadata of the scanned files, the **repository manager**, a component for storing and managing these metadata, the **metadata representation editor** which is responsible for changing the RDF representations of metadata and the **controller**, a component that controls the entire process and metadata life-cycle. The overall architecture of the system is shown in Figure 5.1. We have adopted a modular design with well-defined interfaces in order to be able to extend or replace a component easily. For instance, we can easily switch to another metadata extractor or use different repository storage approaches (as we discuss later on).

5.3.1 Controller

The tool starts like an AntiVirus program, i.e. it starts scanning all files and subfolders that originate from a certain folder (that is specified by the user). At the first scan a metadata record

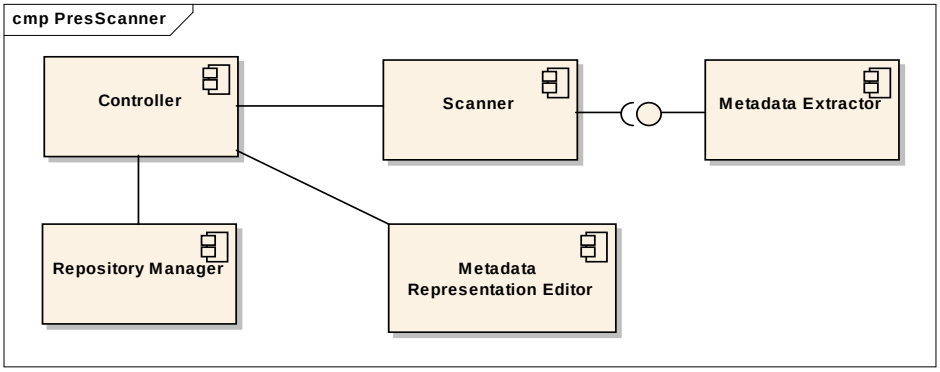

Figure 5.1: The Component diagram of `PreScan`

is created for each encountered file and stored through the Repository Manager. After the end of the scan the user can browse the repository and add extra metadata or edit the extracted ones. The difficult task is to keep the repository consistent while the file system changes. Recall that files are deleted, renamed or change positions and new files are created. `PreScan` uses the full pathname of a file as its identity. Let us consider the case where a file is renamed. For sure we would not like to delete the old metadata record of that file and create a new one since we would lose all metadata that could have been entered manually by the user. If at each scanning we keep a copy image of the entire filesystem then at each subsequent scan we can compare the contents of each encountered file with the contents of the copied files of the previous scan in order to identify file name/position changes. However to keep a copy of the entire file system would be space consuming and the comparison would be unacceptably slow. To overcome this problem we compute and store the *md5 checksum* of the contents of every scanned file inside its metadata record, which is typically expressed as a fixed size hexadecimal number (32 digits). If a file does not change over time, then its md5 will remain intact. Obviously the md5 of a file should be updated whenever a file changes. We deal with file movements in a similar way.

The re-scanning process consists of two phases: the *scanning phase* and the *integration phase*. At the first phase the algorithm scans the filesystem. At the second phase (integration phase) the system tries to identify file additions, modifications and deletions based on the contents signatures (in our case md5), and asks from the user to verify the identified events.

Figure 5.2 sketches the algorithm of rescan. At first we retrieve the list of scanned files and for every file we encounter we compare it with the list of the previously scanned files to decide weather this file existed during the previous scan or it is a new file. If a file existed in

the previous scan and has been changed since the last scan, we extract its new metadata and update the repository appropriately. *PendingList* is used for keeping the files that were not present at the previous scan (obviously these files are not associated with any metadata record). Let sf be such a file. This means that either: (a) sf is a new file, or (b) sf was moved from another folder, or (c) sf is an old file that has been renamed and `PreScan` cannot recognize it. In contrary, *ObsoleteList* keeps metadata records that no longer correspond to a file which means that either the specified file has been removed or the file has been moved to another location. At a final step we recognize file movements and removals by comparing the content of the files in the *PendingList* and *ObsoleteList* lists.

Algorithm **ReScan**()
Input: Rep, Scanner, Extractor
Output: updated repository

1. PendingList = ObsoleteList = $\emptyset$
2. PreviousFiles = Rep.getAllFiles($ScannerConf$)
3. CurFiles = Scanner.getScannedFiles($ScannerConf$)
4. for each file f in CurFiles
5. if $f \in$ PreviousFiles //f *existed in the previous scan*
6. if f.lastModified > Rep.getLastModified(f.path) //f *has changed*
7. m=Extractor.extractMetadata(f.path)
8. Rep.update(f.path,m)
9. else //f *is a "new" file*
10. PendingList = PendingList $\cup\{f\}$
11. end for
12. $ObsoleteList$ = PreviousFiles $\setminus$ CurrFiles
13. for each file f in $PendingList$
14. m=Extractor.extractMetadata(f.path)
15. of = ObsoleteList.getMatched(f.content) // *e.g. through MD5*
16. if (of==null) then // *nothing matched*
17. Rep.add(f,m)
18. $PendingList = PendingList \setminus \{f\}$
19. else
20. If UserVerifiesMapping(f, of) then
21. Rep.updateMetadata(of, f,m)
22. $PendingList = PendingList \setminus \{f\}$
23. $ObsoleteList = ObsoleteList \setminus \{of\}$
24. end for
25. Rep.DeleteOldRecords($ObsoleteList$)

Figure 5.2: The algorithm of `PreScan`

We have developed a GUI to aid users in establishing mappings between the elements of the *pending* and the *obsolete* list. Figure 5.3 shows an indicative screendump. The upper part of the

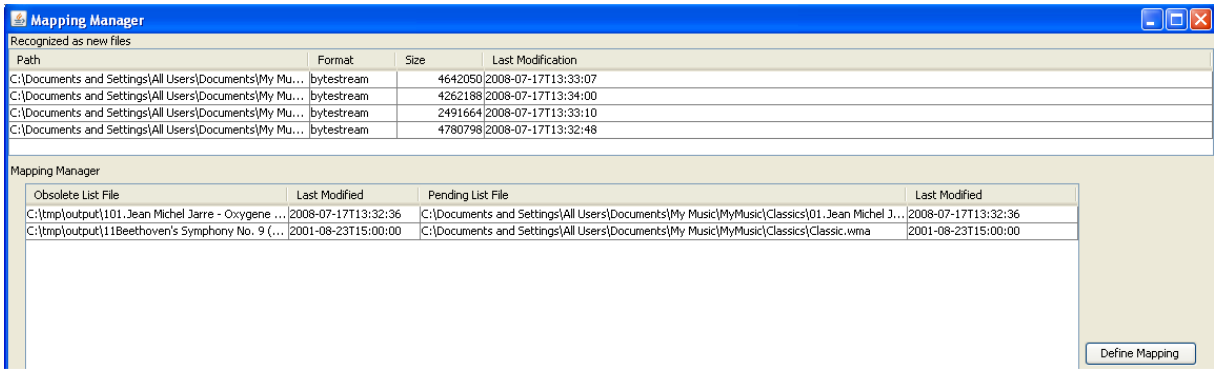

Figure 5.3: The GUI for managing mappings

window contains the new files that were found during the re-scanning and the bottom part all possible mappings from files that were moved/renamed. In this particular case we have 4 new files, 1 file movement and 1 file that was renamed.

5.3.2 Metadata Extractor

For every file `PreScan` extracts and keeps its filetype, path, owner, last modification date and size. Moreover we extract and keep fileformat-specific embedded metadata. `PreScan` uses JHOVE as a format detector and metadata extractor. It recognizes several formats (AIIF, ASCII, BYTESTREAM, GIF, HTML, JPEG, JPEG 2000, PDF, TIFF, UTF-8, WAVE, XML) and Table 5.2 shows some of the extracted metadata while Figure 5.4 shows a part of the JHOVE XML output schema. The element *repInfo* contains a subelement *property* that contains name-value pairs with technical information about a file (i.e. for images it will contain ColorSpace information, ExposureMode information, resolution etc).

5.3.3 Repository Manager

The Repository Manager is responsible for storing and updating the metadata records. The metadata record of a file includes both the extracted and the human-provided metadata. There are more than one choices regarding where these metadata are stored. The options that are currently supported are listed below (they are not mutually exclusive):

- (SF) For each scanned file its metadata record is created and stored in a Specific Folder specified by the user.

Table 5.2: Recognized Formats and Extracted Metadata from JHOVE

Format	Extracted Metadata
ALL Formats	ScanDate, FilePath, LastModificationDate, Size, Format, Status, Module, MimeType
AIFF	AudioDataEncoding, ByteOrder, FirstSampleOffset, Hours, Minutes,Seconds, Sample, NumberOfSamples
ASCII	LineEndings
GIF	GraphicRenderingBlocks, LogicalScreenWidth, LogicalScreenHeight, ColorResolution, BackGroundColorIndex, PixelAspectRatio, CompressionScheme, TransparencyFlag, ImageWidth, ImageLength, BitsPerSample
HTML	PrimaryLanguage, OtherLanguages, Title, MetaTags (name, HttpEquiv, Content), Frames (name, Title, LongDesc, Src), Links, Scripts, Images(Alt, Longdesc, Src, Height, Width) , Citations, DefinedTerms, Abbreviations(Text, Title), Entities, UnicodeEntityBlocks
JPEG	CompressionType, ScanerManufacturer, ScannerModuleName, ImageWidth, ImageLength, BitsPerSample, SamplesPerPixel, PixelAspectRatioX, PixelAspectRatioY, Precision, ColorSpace, PixelXdimension, PixelYdimension, DateTimeOriginal, DateTimeDigitized, ExposureTime, LigthSource, Flash, FileSource, SceneType, Saturation, Sharpness
JPEG 2000	Brand, MinorVersion, Compatibility, precedence, XSize, YSize, BitsPerSample, SamplesPerPoxel, Creator
PDF	PageLayout, PageMode, Creator, Producer, CreationDate, Fonts
TIFF	ByteOrder, CompresiionScheme, SamplingFrequency, XSamplingFrequency, YSamplingFrequency, ImageWidth, ImageLength, BitsPerSample, SamplesPerPixel
UTF-8	LineEndings, Additional Control Characters, NumberOfCharacters, UnicodeCodeBlocks
WAVE	SchemaVersion, AudioDataEncoding, FrameCount, TimeBase, videoField, CountingMode, Hours, Minutes, Seconds,Frames, SampleRate, NumberOfSamples, NumChannels
XML	Version, Encoding, StandAlone, DTD,(publicID, SystemID, InternalSubset) Schemas,(Schema (namespaceURI, SchemaLocation)) Root, NameSpaces, (NameSpace (Prefix, URI)) Notations, (Notation (Name, PublicID, SystemID)) CharacterReferences, Entities, (Entity (Name, Type, Value, PublicID, SystemID, Notation)) ProcessingInstructions, (ProcessingInstruction (Target, Data)) Comments (Comment)

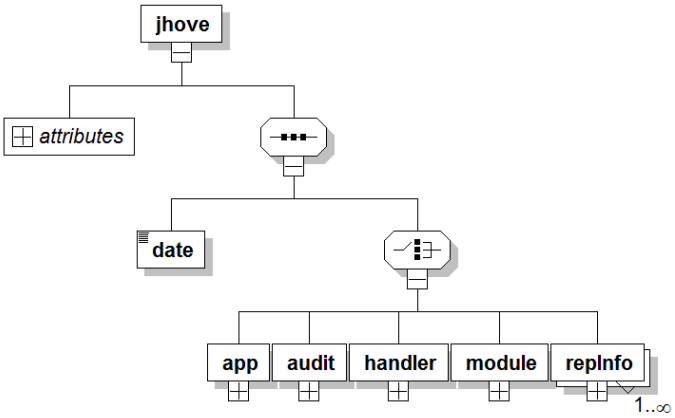

Figure 5.4: A fragment of the JHOVE output XML schema

- (OF) For each scanned file its metadata record is created and stored in the same folder with the Original scanned file.
- (KB) The contents of the metadata records of scanned files are stored in a Semantic Web-based Knowledge Base. In that case the Repository Manager also offers querying services.

The extracted metadata of a digital file that are stored using the repository manager and can be later viewed or enriched by the user. For example assume that `PreScan` scans a file named `forth20090115.jpg`. It extracts various metadata about that file (e.g. image resolution, date, information about the digital camera captured this photo etc.) and stores them in the repository. Later on the user can enrich these metadata by adding human provided metadata such as a description for that photograph (e.g. "FORTH cake event in 15-01-2009"). Now suppose that this file is moved to another location. At the next scan, `PreScan` will identify this change and will update its metadata record with the new file location. Therefore the (user-provided) metadata for that file will not get lost. In case the KB option has been adopted, the user can also search for a file by querying the repository according to the extracted information. For example we could search for a photograph with description about "FORTH cake event" or photographs taken by a specific digital camera, etc.

Metadata are extracted from JHOVE and are stored in XML format. These files can be stored in a XML database and then queried using XQuery. Instead of having an XML-based framework an alternative approach is to define an ontology that allows expressing all the extracted metadata and can exploit the inheritance semantics of RDF/S.

To this end we decided to exploit the CIDOC CRM and its extensions. Figure 5.5 shows the general architecture of ontologies that we propose. On top we have the CIDOC CRM [21] ontology that offers the basic conceptual modeling, in the middle layer there is the $\texttt{CRM}_{dig}$ ontology which is adequate for capturing the semantics of digital objects as well as their provenance, and at the bottom layer we have `COD` ontology that is used for expressing dependencies. Finally we have any domain specific extensions. The instance layer will contain the automatically extracted metadata, the manually provided metadata and the `COD`-based descriptions.

This view enables the integration of different information about digital objects (i.e. provenance, dependencies). For example the `COD`-based description can capture information about the file formats and software modules. In this way information about the scanned files can be linked with the descriptions of these formats. Specifically every scanned file is defined as a module (is assigned a unique module identifier), and dependencies regarding the file format and

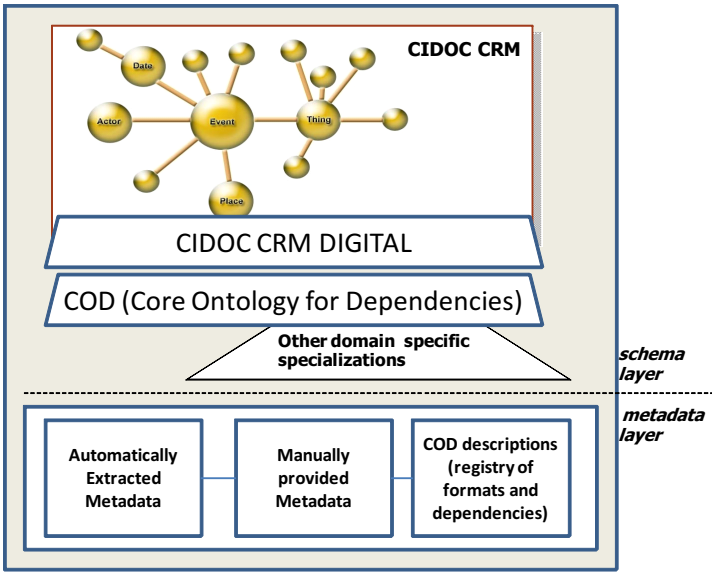

Figure 5.5: Architecture of Semantic Web Ontologies and Metadata

the appropriate software are defined automatically based on their type (i.e. `PreScan.jpeg` > `JPEG`, `PreScan.jpeg` $>_{render}$ `Image Viewer`), but the user can easily add more dependencies or edit the existing ones using `GapMgr` (Section 3.4.2).

Regarding the transformation from XML to RDF, `PreScan` parses the XML metadata records (the output of JHOVE + human provided metadata) and produces instances according to CIDOC CRM and its extensions. More precisely we have predefined a set of XSLT transformations that maps metadata to ontological instances according to the above ontologies. Every different file type is defined as a subclass of `C1 Digital Object` which is the class for representing digital objects from $\texttt{CRM}_{dig}$ (Chapter 4). Table 5.3 describes how various metadata are transformed to ontological instances according to CIDOC CRM and its extension. For each metadata the path over the semantic graph denotes how it is modeled. For example the md5 checksum of a file will be an instance of `E41 Appellation` and will be associated with an instance of digital object (representing the file) through the *P1F is identified by* property.

5.3.4 Metadata Representation Editor

However someone may find out that a path (in the semantic web graph) that is used for representing a metadata could be represented using another path (from the same or from another ontology). Additionally someone may create his own ontology and therefore define the RDF

Table 5.3: Metadata According to CIDOC CRM

FileType	Metadata	Path
All	Scan Date	`C1 Digital Object` → *S2B was source for* → `C10 Software Execution` → *P4F has time span* → `E52 Time-Span` → *P80F end is qualified by*
All	Last Modification Date	`C1 Digital Object` → *S11B was output of* → `C26 Digital Machine Modification Event` → *P4F has time − span* → `E52 Time-Span` → *P80F end is qualified by*
All	File Size	`C1 Digital Object` → *P43F has dimension* → `E54 Dimension` → (*P91 has unit* → `E58 Measurement unit = "bytes"`) → *P90F has value* → `E60 Number`
All	MD5 Checksum	`C1 Digital Object` → *P1F is identified by* → `E41 Appellation`
Image	Height	`C1 Digital Object` → *P43F has dimension* → `E54 Dimension` → (*P91F has unit* → `E58 Measurement unit = "pixels"`) → *P90F has value*
Sound	Duration (sec.)	`C1 Digital Object` → *P43F has dimension* → `E54 Dimension` → (*P91F has unit* → `E58 Measurement unit = "seconds"`) → *P90F has value*
HTML	Links	`C1 Digital Object` → *P148F has component* → `E73 Information Object` → (*P3F has note* = *"anchorText"*) → (*P67F refers to* → `E73 Information Object`)
HTML	Title	`C1 Digital Object` → *P102F has title* → `E35 Title`

representations according to it. For this purpose we implemented a small tool (embedded in `PreScan`) that aid users changing the RDF representations of metadata. Indicatively Figure 5.6 shows the metadata editor window for editing the representation of the file size of HTML files. The right part contains the current representation in a text window which allow users add or delete RDF code. The left part of the window is used to aid users with the process of editing. More specifically the upper frame contains all the classes of the selected ontology (here the selected ontology is CIDOC CRM) and the lower frame contains the available properties. The selection of a class from the upper part will restrict the set of available properties by showing only the available ones (direct and inherited properties). Apart from the predefined ontologies that are embedded in metadata editor the user can also define a RDF representation from another ontology following the same procedure as above. RDF representations can be checked for validity using the validation mechanism of the SWKM-MainMemoryModel2 [5]. The RDF export functionality will eventually transform metadata according to the new representations.

5.3.5 Evaluation of `PreScan`

Regarding efficiency, the extraction of the embedded metadata depends on the type and the size of the file. In general, the bigger the size of a file is, the more time it takes to be

[5] http://athena.ics.forth.gr:9090/SWKM

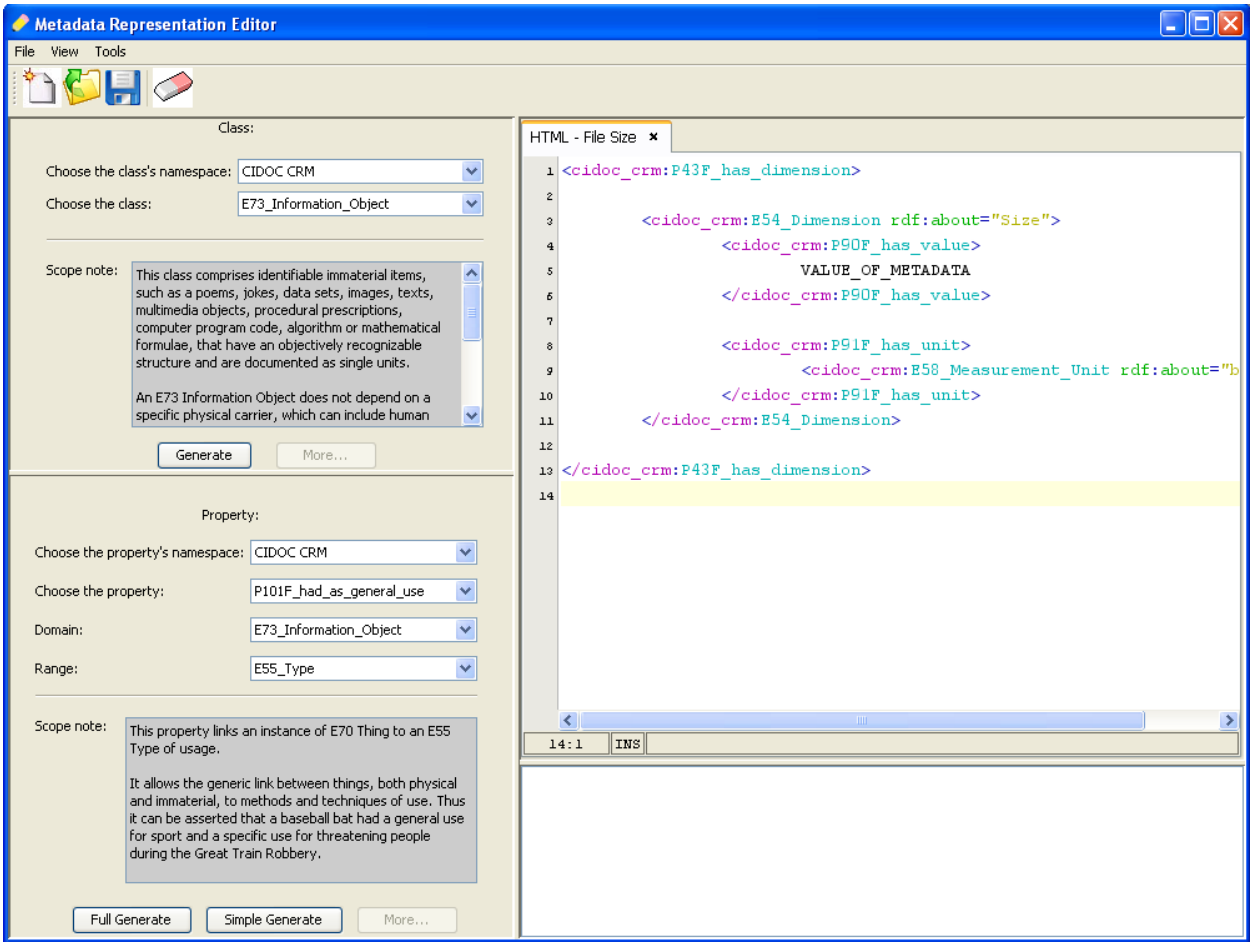

Figure 5.6: Editing the RDF representation of metadata

scanned. Unknown filetypes (i.e. filetypes that are not recognized by JHOVE) are identified as BYTESTREAM modules and only the basic metadata (1^{st} row in Table 5.2) are extracted.

We performed some experiments over various datasets. Table 5.4 summarizes the results of the evaluation. For every dataset we report the total time for the scanning process as well as the time for every subtask (extract and store times). Additionally we report the time for the transformation of metadata records to CIDOC CRM instances. For every dataset we also report the number of the different files comparing to the total files of the set. The experiments were done in a 32-bit system with INTEL Core 2 Duo processor at 2.00 GHz, using 2 GB of RAM running Windows 7. The average size of a file in each dataset was approximately 800 KB.

Table 5.4: Time Performance of the Scanning process

DataSets			**Time (sec)**			
Files	Type		Extract	Store	Total	RDF Transform
10	html/xml wave/aiff ascii/utf8 jpeg/gif pdf unknown	2 2 1 3 1 1	0.75	0.1	0.86	1.23
10^2	html/xml wave/aiff ascii/utf8 jpeg/gif pdf unknown	19 11 7 36 8 19	7.3	0.35	7.7	3.52
10^3	html/xml wave/aiff ascii/utf8 jpeg/gif pdf unknown	201 115 56 251 103 274	72.5	4.02	78.34	11.66
10^4	html/xml wave/aiff ascii/utf8 jpeg/gif pdf unknown	2123 1096 922 4629 594 636	814 ($\sim$ 13.5 min)	29	845 ($\sim$ 14.1 min)	107
10^5	html/xml wave/aiff ascii/utf8 jpeg/gif pdf unknown	34069 1027 25467 13478 1240 24539	10256($\sim$ 2 hr. 51 min)	335($\sim$ 5.5 min)	10773($\sim$ 3 hr)	449($\sim$ 7.5 min)

As we can the more time-consuming task is the metadata extraction. During the extraction of the embedded metadata the MD5 checksum of every file is also computed. Clearly this task is independent from the repository mode (Section 5.3.3). However the storage time depends on the repository mode adopted. Here we have used the SF option and stored the metadata records as XML documents. We have noticed that the entire process takes about 3 hours for

100 thousand files (about 80 GB).

5.4 Related Approaches

There are only few related works. For instance, the work presented in [19, 18] scans the filesystem, extracts the type of the digital files and detects those files having obsolete types. Its functionality relies on three registries: a software version registry, a format registry, and a recommended format registry.

Another approach based on migration is described in [13]. It adopts a SOA (Service-Oriented Architecture) for enabling the combination of different format detectors and registries to support automatic migration. The key difference with our work is that we extract the embedded metadata, we link them with data that enable dependency management services and we focus on supporting the entire life-cycle of metadata (i.e. the automatic identification of file removals and movements).

[36, 35] presents an approach for web page preservation. It aims at enabling the web server as a just-in-time metadata-extractor/generator. The archivist (or client) receives the metadata of a web page at dissemination time (as an XML-formatted response), and of course this adds an extra overhead for the web-server. That work pre-supposes that all digital files are placed in a web server and that the extended web-server has been installed.

Empirical Walker [3] is probably the most similar tool with `PreScan`. It also scans the file system, it determines file formats, it analyzes file contents calculates checksums, and associates external metadata. It also adopts JHOVE as metadata extractor. Regarding file format identification, Empirical Walker first assigns a MIME-type to every file according to a mapping table (that maps file extensions to MIME-types) and in a later phase it uses JHOVE to validate the associated MIME-types and extract more technical metadata (only for those files that are supported from JHOVE). However associating MIME-types based only on the file-extensions is error-prone because file extensions are often unreliable or missing. Furthermore, and in comparison to our work, we allow the addition of user-provided metadata we can output the resulting metadata in various ways and formats also exploiting the expressive power of Semantic Web languages. Specifically the resulted metadata can be access or edited: (a) through the developed GUI, (b) in txt format (in a human readable format), (c) in XML which is directly output from JHOVE and (d) through a SW repository.

Metadata Miner Catalogue[6] is a commercial software tool that lists files and folders summary information, extracts file properties and their metadata and create reports in various formats (including XML, HTML and RDF according to Dublin Core schema). Furthermore it identifies several file formats but does not recognize the specific version of that format. For example it will recognize a .doc file as a Microsoft Word document but cannot identify whether it is a Microsoft Word 6 document or a Microsoft Word 2003 document. Additionally its free version allows the listing of only ten files per folder and extracts some of the available metadata.

TripFS [34] is a Java-based server software that crawls the file system starting from a given directory, extracts a limited set of low-level metadata (i.e. name, size, owner, creation date, etc.), export them as RDF descriptions, links file resources to other relevant data sources (i.e. it links a paper in PDF with the DBLP collection) and exposes these data sets according to Linked Data principles. Moreover it identifies file movements and deletions. However it does not allow the manual addition of metadata. Additionally after a file movement is identified the mapping with its description is done using a set of heuristics without any confirmation by the user, which may lead to inconsistencies.

Table 5.5: Comparing `PreScan` with related systems

	`PreScan`	EW	Droid	ME	MMC	TripFS
ReScan with preservation of manual metadata	✓					
Identification of file movements	✓					✓
Mapping Confirmation by the user	✓					
Export to RDF	✓				✓	✓
Exploitation of format registries	✓(Pronom + Dependencies)		✓(Pronom)			
Format identification	✓	✓	✓	✓^{n1}	✓^{n1}	✓
Compliance with dependency management	✓	✓^{n2}				

^{n1}: Identifies format but no details about the particular version.
^{n2}: Structural dependencies are extracted (however no details are available).

Table 5.5 compares our work (`PreScan`) with other tools like Empirical Walker **(EW)**, DROID, Metadata Extractor **(ME)**, Metadata Miner Catalogue **(MMC)** and TripFS. Notice that `PreScan` is the only tool that allows file system re-scans that protect the human-provided

[6] http://peccatte.karefil.com/software/Catalogue/MetadataMiner.htm

metadata, and identifies file movements.

5.5 Summary

In this chapter we described the design and implementation of `PreScan`, a tool for automating the ingestion, maintenance and transformation of metadata of digital files. `PreScan` scans a filesystem, extracts the embedded metadata of files, binds them with manually provided, and supports processes for ensuring the freshness of the metadata repository without loosing the human provided metadata. Besides it transforms metadata to ontological descriptions according to CIDOC CRM and its extensions and also supports the efficient alteration of the ontological representation that is used. The architecture of ontologies that is used from the Semantic Web-based repository ensures interoperability and enables the provenance information exchange and dependency management services that assist the preservation of intelligibility of digital objects.

The latest version of `PreScan` (Beta 1.1) is available for download and use [7].

[7] http://www.ics.forth.gr/prescan/

Chapter 6

Conclusions and Future Work

The preservation of digital objects is critically dependent on the successful preservation of their viability, renderability, understandability etc. (recall Section 1.1). In this thesis we concentrate on the preservation of the intelligibility and provenance of digital objects. Specifically we described some models and tools for the preservation of digital objects.

We proposed a model for the preservation of intelligibility, based on dependency management. This perspective allows us to answer queries of the form: (a) what kind of representation information do we need , (b) how this depends on the designated community and (c) what kind of automation can we offer (regarding packaging and dissemination). We also discussed about modeling and implementation frameworks for the realization of this model based on the different semantics of dependencies.

Additionally we addressed the need for an extensible conceptual framework that will allow provenance information to be integrated, exchanged and exploited within or across digital archives. To this end we extended CIDOC CRM, defining CIDOC CRM Digital and showed how it can be used for querying provenance.

Besides we designed and implemented the following tools for digital preservation:
(a) `GapMgr`: A tool that can aid several tasks related to the preservation of intelligibility.
(b) `PreScan`: A tool that scans a filesystem, automatically extracts the embedded metadata, enrich them with user-defined metadata, transforms them to instances according to CIDOC CRM and its extensions, and supports processes for ensuring the freshness of the metadata repository.

The completeness of this work can be justified from (a) the fact that it was founded on the

conceptual model CIDOC CRM [21], which is an ISO standard (ISO 21127:2006), (b) its compliance with the reference model of OAIS [20], which is also an ISO standard (ISO 14721:2003), and (c) its applicability in the context of the project CASPAR [1] using real data.

One issue that is worth for further research is the extension of the model for the preservation of intelligibility with converters. Conversion (or transformation) is becoming a common practice for achieving interoperability, so it could be exploited for the problem at hand. Additionally we should remark that further testing of the models and tools is an open ended process for the future.

Bibliography

[1] CASPAR (Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval), FP6-2005-IST-033572 (http://www.casparpreserves.eu/).

[2] A. Ames, N. Bobb, S.A. Brandt, A. Hiatt, C. Maltzahn, E.L. Miller, A. Neeman, and D. Tuteja. Richer File System Metadata Using Links and Attributes. In *Proccedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 49–60, Monterey, CA, USA, April, 2005.

[3] R. Anderson, H. Frost, N. Hoebelheinrich, and K. Johnson. The AIHT at Stanford University: Automated Preservation Assessment of Heterogeneous Digital Collections. *D-Lib Magazine*, 11:12, December, 2005.

[4] M. Belguidoum and F. Dagnat. Dependability in Software Component Deployment. In *Proceedings of the 2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'07)*, pages 223–230, Szklarska Poreba, Poland.

[5] M. Belguidoum and F. Dagnat. Dependency Management in Software Component Deployment. *Electronic Notes in Theoretical Computer Science*, 182:17–32, 2007.

[6] H. Christiansen and V. Dahl. HYPROLOG: A New Logic Programming Language with Assumptions and Abduction. In *Proceedings of the 21st Conference on Logic Programming (ICLP'05)*, pages 159–173, Sitges, Barcelona, Spain, October, 2005.

[7] H. Christiansen and V. Dahl. Assumptions and Abduction in Prolog. In *Proceedings of the 3rd International Workshop on Multiparadigm Constraint Programming Languages (MultiCPL'04); Proceedings of the 20th International Conference on Logic Programming (ICLP'04)*, pages 87–101, Saint Malo, France, September, 2004.

[8] L. Console, D.T. Dupre, and P. Torasso. On the Relationship Between Abduction and Deduction. *Journal of Logic and Computation*, 1(5):661–690, 1991.

[9] S.B. Davidson and J. Freire. Provenance and Scientific Workflows: Challenges and Opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1345–1350, Vancouver, Canada, June, 2008.

[10] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L.A. Stein. OWL Web Ontology Language 1.0 Reference, July, 2002. (http://www.w3c.org/TR/owl-ref).

[11] XFDU development site. (http://sindbad.gsfc.nasa.gov/xfdu).

[12] T. Eiter and G. Gottlob. The Complexity of Logic-based Abduction. *Journal of the ACM (JACM)*, 42(1):3–42, January, 1995.

[13] M. Ferreira, A.A. Baptista, and J.C. Ramalho. A Foundation for Automatic Digital Preservation. *Ariadne*, 48, July, 2006.

[14] M.S. Fox and J. Huang. Knowledge Provenance: An Approach to Modeling and Maintaining the Evolution and Validity of Knowledge. 2003.

[15] X. Franch and N.A.M. Maiden. Modeling Component Dependencies to Inform their Selection. In *Proceedings of the 2nd International Conference on COTS-Based Software Systems (ICCBSS'03)*, Ottawa, Canada, February, 2003.

[16] J. Freire, D. Koop, E. Santos, and C.T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science and Engineering*, 10(3):11–21, May-June, 2008.

[17] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, May 2004. (http://www.w3.org/Submission/SWRL/).

[18] J. Hunter and S. Choudhury. A semi-automated digital preservation system based on semantic web services. In *Proceedings of the 4th ACM/IEEE-CS joint Conference on Digital Libraries (JCDL '04)*, pages 269–278, New York, NY, USA, 2004.

[19] J. Hunter and S. Choudhury. PANIC: an Integrated Approach to the Preservation of Composite Digital Objects Using Semantic Web Services. *International Journal on Digital Libraries*, 6(2):174–183, April, 2006.

[20] International Organization For Standardization. OAIS: Open Archival Information System – Reference Model, 2003. Ref. No ISO 14721:2003.

[21] International Organization For Standardization. The CIDOC Conceptual Reference Model, 2006. Ref. No ISO 21127:2006 (http://cidoc.ics.forth.gr/).

[22] M. Jarrar and R. Meersman. Formal Ontology Engineering in the DOGMA Approach. In *Proceedings of the International Conference on Ontologies, Databases and Applications of Semantics (ODBase'02)*, pages 1238–1254, Irvine, California, USA, October, 2002.

[23] AC Kakas, RA Kowalski, and F. Toni. The Role of Abduction in Logic Pro-gramming. *Handbook of Logic in Artificial Intelligence and Logic Programming: Logic programming*, page 235, 1998.

[24] G. Karvounarakis, V. Christophides, and D. Plexousakis. Querying Semistructured (Meta)data and Schemas on the Web: The case of RDF & RDFS. Technical Report 269, ICS-FORTH, 2000. Available at: http://www.ics.forth.gr/proj/isst/RDF/rdfquerying.pdf.

[25] DEDSL Language. (http://east.cnes.fr/english/index.html).

[26] EAST Language. (http://east.cnes.fr/english/page_east.html).

[27] R.A. Lorie. Long Term Preservation of Digital Information. In *Proceedings of the 1st ACM/IEEE-CS joint Conference on Digital Libraries*, pages 346–352, Roanoke, Virginia, USA, 2001.

[28] A. Lucas. XFDU Packaging Contribution to an Implementation of the OAIS Reference Model. In *Proceedings of the International Conference PV'2007 (Ensuring the Long-Term Preservation and Value Adding to Scientific and Technical Data)*, Edinburgh, November 2005.

[29] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. RUL: A Declarative Update Language for RDF. In *Proceedings of the 4th International Conference on the Semantic Web (ISWC-2005)*, Galway, Ireland, November 2005.

[30] Y. Marketakis, M. Tzanakis, and Y. Tzitzikas. PreScan: Towards Automating the Preservation of Digital Objects. In *Proceedings of the International Conference on Management of Emergent Digital EcoSystems (MEDES'09)*, pages 404–411, Lyon, France, October, 2009.

[31] A. Mikroyannidis, Ong Bee, Kia Ng, and D. Giaretta. Ontology-based Temporal Modelling of Provenance Information. In *Proceedings of the 14th IEEE Mediterranean Electrotechnical Conference, MELECON 2008*, pages 176–181, Tenerife, Spain, May 2008.

[32] L. Moreau, J. Freire, J. Myers, J. Futrelle, and P. Paulson. The Open Provenance Model v1.1. *University of Southampton*, December, 2009.

[33] GDFR (Global Digital Format Registry). (http://www.gdfr.info).

[34] B. Schandl and N. Popitsch. Lifting File Systems into the Linked Data Cloud with TripFS. In *Proceedings of the WWW 2010 Workshop Linked Data on the Web (LDOW'10)*, Raleigh, North Carolina, April, 2010.

[35] J.A. Smith and M.L. Nelson. A Quantitative Evaluation of Dissemination-time Preservation Metadata. In *Proceedings of the 12th European conference on Research and Advanced Technology for Digital Libraries (ECDL'08)*, pages 346–357, Aarchus, Denmark, 2008. Springer.

[36] J.A. Smith and M.L. Nelson. Creating preservation-ready web resources. *D-Lib Magazine*, 14(1/2):1082–9873, 2008.

[37] H. Stenzhorn, K. Srinivas, M. Samwald, and A. Ruttenberg. Simplifying Access to Large-Scale Health Care and Life Sciences Datasets. *Lecture Notes in Computer Science*, 5021:864, 2008.

[38] E. Sunagawa, K. Kozaki, Y. Kitamura, and R. Mizoguchi. An Environment for Distributed Ontology Development Based on Dependency Management. In *Proceedings of the 2nd International Semantic Web Conference (ISWC'03)*, pages 453–468, Sanibel Island, Florida, USA, October, 2003. Springer.

[39] The technical registry PRONOM (The National Archives). (http://www.nationalarchives.gov.uk/pronom).

[40] M. Theodoridou, Y. Tzitzikas, M. Doerr, Y. Marketakis, and V. Melessanakis. Modeling and querying provenance by extending CIDOC CRM. *Journal of Distributed and Parallel Databases*, 27:169–210, 2010.

[41] Y. Tzitzikas and G. Flouris. Mind the (Intelligibility) Gap. In *Proceedings of the 11th European Conference on Research and Advanced Technology for Digital Libraries (ECDL'07)*, Budapest, Hungary, September 2007.

[42] Y. Tzitzikas, D. Kotzinos, and Y. Theoharis. On Ranking RDF Schema Elements (and its Application in Visualization). *Journal of Universal Computer Science*, 13(12):1854–1880, 2007.

[43] Y. Tzitzikas and Y. Marketakis. Automating the ingestion and transformation of embedded metadata. *ERCIM News*, 2010(80), 2010.

[44] Y. Tzitzikas, Y. Marketakis, and G. Antoniou. Task-based Dependency Management for the Preservation of Digital Objects using Rules. In *Proceedings of the 6th Hellenic Conference on Artificial Intelligence (SETN'10)*, Athens, Greece, May, 2010.

[45] J.R. van der Hoeven, R.J. van Diessen, and K. van der Meer. Development of a Universal Virtual Computer (UVC) for long-term preservation of digital objects. *Journal of Information Science*, 31(3):196, 2005.

[46] M. Vieira, M. Dias, and D.J. Richardson. Describing Dependencies in Component Access Points. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 115–118, Toronto, Canada, May, 2001.

[47] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. In *Proccedings of the 17th IEEE International Conference on Automated Service Engineering, ASE'02*, Los Alamitos, CA, USA, 2002.

[48] M. Walter, C. Trinitis, and W. Karl. OpenSESAME: an Intuitive Dependability Modeling Environment Supporting Inter-Component Dependencies. *Proceedings of Pacific Rim International Symposium on Dependable Computing*, pages 76–83, 2001.

Index