The original version of the chapter: Y. Tzitzikas, Y. Marketakis, V. Christophides, Preservation of Intelligibility of Digital Objects,in D.G. Giarretta ed. Advanced Digital Preservation, Springer 2011

1 Preservation of Intelligibility of Digital Objects

1.1 On Digital Objects and Dependencies

We live in a digital world. Everyone nowadays works and communicates 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. in paper) is now 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 the 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 can 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 into two broad categories based on the different interpretations of their content: information objects and computational objects. Information objects encode knowledge about something in a data structure that allows their exchangeability. Examples of information objects are documents, data-products, images, ontologies. The contents of information objects can many times be described straightforwardly in non digital form. For example, for a reader's eyes the contents of a digital document as rendered in a computer screen are the same with the contents of a printout of that document. 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. To understand them we may need extra information which can be expressed using other information objects. For example, to understand the concepts of an ontology A we have to know the concepts defined in each ontology B that it is used by A. As another example consider scientific data products. Such products are usually aggregations of other (even heterogeneous) primitive data. The provenance information of these data products can be preserved using scientific workflows [1]. The key advantage of scientific workflows is that they record data (and process) dependencies during workflow runs. For example, Figure 1 shows a workflow that generates two data products. The rectangles denote data while the edges capture derivations and thus express the dependability of the final products from the initial and intermediate results. Capturing such dependencies is important for the understandability, reproducibility and validity of the final data products.

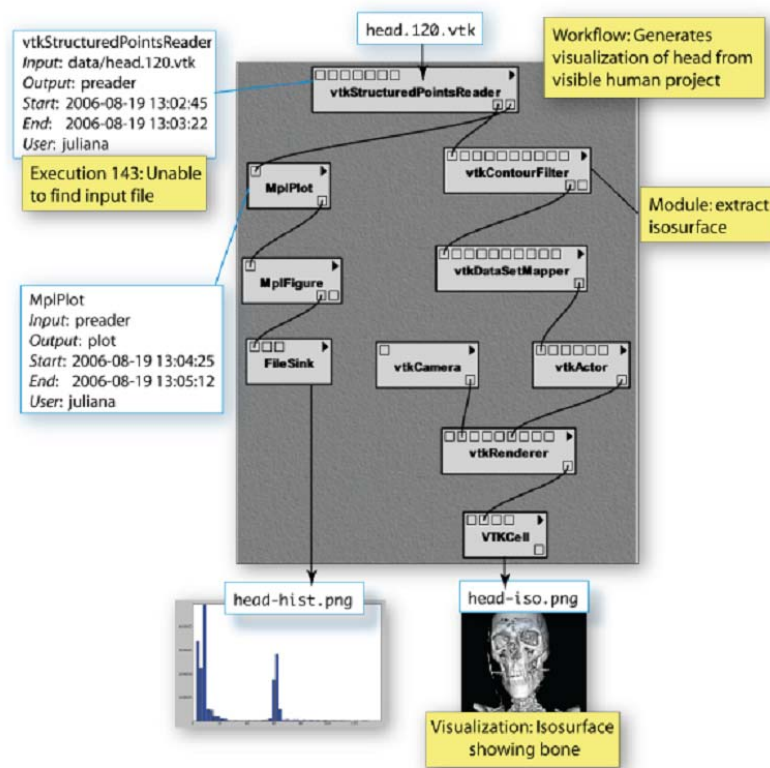

Figure 1: The generation of two data products as a workflow

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 performs complex mathematic calculations does not explicitly define the memory addresses that will keep the numbers. Instead it exploits the functionalities provided by other programs that handle such 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 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, e.g. we cannot run the above java program if we haven't installed a Java Virtual Machine.

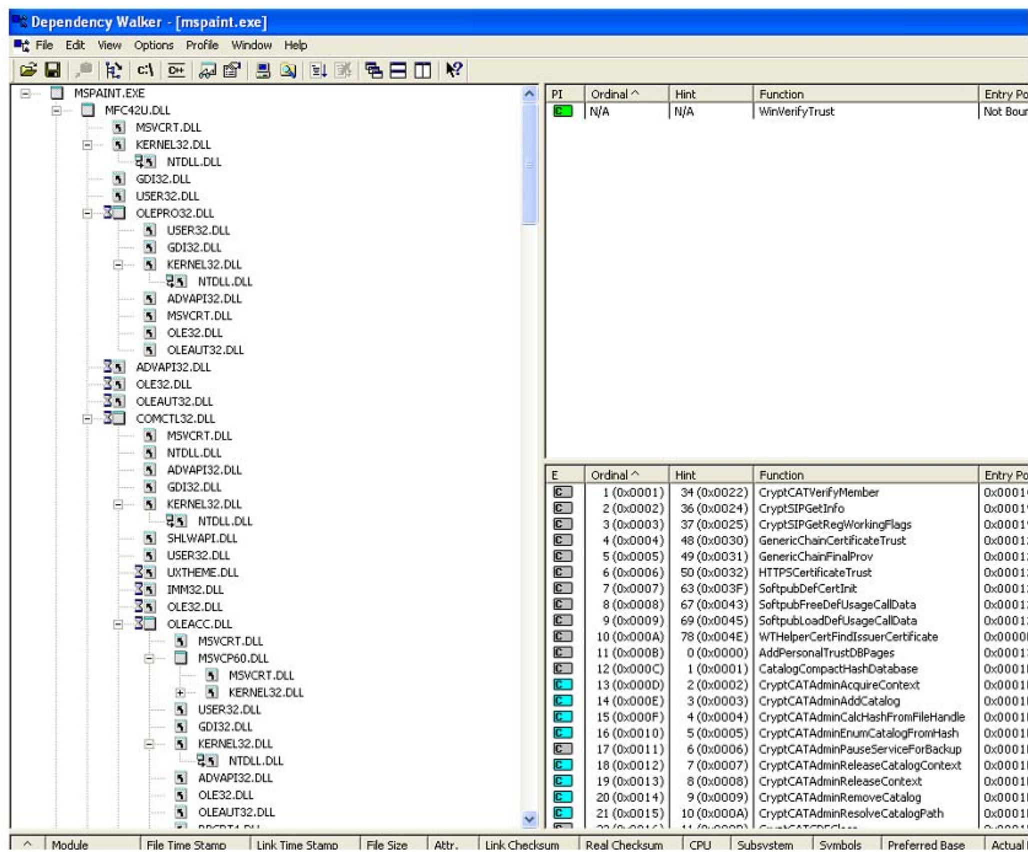

Figure 2: The dependencies of mspaint software application

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 [2][3] dependencies between information objects are exploited for ensuring consistency and validity. Figure 1 illustrates dependencies over the data products of a scientific workflow and such dependencies apart from being useful for understanding and validating the data, they can also allow or aid the reproduction of the final data products. [4] proposes an extended framework for file systems that allows users to define dependency links between files where the semantics of these links are defined by users through key-value pairs. Finally, many dependency management approaches for software components [5][6][7][8][9] have been described in the literature. The interpretation of software dependencies varies and concerns the installation or de-installation safety, the ability to perform a task, the selection of the most appropriate component, or the consequences of a component's service call to other components.

1.1.1 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*. According to

OAIS, 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 the text. The information in a digital text file 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. This kind of metadata is also important for packaging. OAIS distinguishes packages to AIPs (Archival Information Packages) and DIPs (Dissemination Information Packages), that are derived from one or more AIPs as a response to a request of an OAIS. Packaging is analyzed in more detail at Section 1.2.3.3. Every Data Object needs a RI object that interprets it and this relationship (*interpretedUsing*) is actually a specialized form of dependency between Data 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 is a responsibility of an OAIS to find (or create new) software conforming to the given Representation Information for this information object.

For example a file in FITS format is understandable only from persons that know how to handle this format. Someone who does not understand this file needs additional Representation Information. Figure 3 shows such a representation network for a FITS file. Rectangles denote RI objects while edges 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 inside the FITS files. Consequently, the RI for FITS includes the FITS Documentation and the FITS Dictionary which are expressed in PDF and XML respectively. In turn, their interpretation requires extra RI and so on.

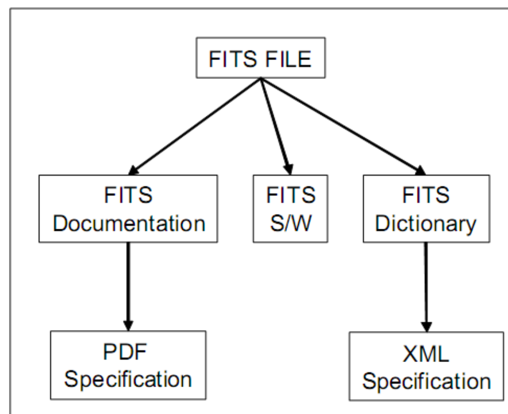

Figure 3: The Representation network of a FITS file

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.

1.2 A Formal Model for the Intelligibility of Digital Objects

1.2.1 A Core Model for Digital Objects and Dependencies

We introduce a core model for the intelligibility of digital objects based on dependencies. The model has two basic notions: *Module* and *Dependency*. As we described in Section 1.1, digital objects depend on a plethora of other resources. Recall that these resources can be described as objects containing information or using other resources. We use the general notion of ***module*** to model these resources (either information or computational objects). There is no standard way to define a module, so we can have modules of various levels of abstraction, e.g. a module can be a collection of documents, or alternatively every document in the collection can be considered as a module.

In order to ensure the intelligibility of digital objects first we have to identify which are the permissible actions with these objects. This is 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. The reliance of a module on others for the performability of a task is captured using ***dependencies*** between modules. For example for compiling the previous file we can define that it depends on the availability of a java compiler and on all other classes and 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 $\mathcal{T}$ as the set of all modules and the binary relation $>$ on $\mathcal{T}$ is used for representing dependencies. A relationship $t>t'$ ($t,t' \in \mathcal{T}$) means that module t depends on module t'.

The interpretations of modules and dependencies are very general in order to capture a plethora of cases. Specifying dependencies through tasks leads to dependencies with different interpretations. To distinguish the various interpretations we introduce ***dependency types*** and we require every dependency to be assigned at least one type that denotes the objective (assumed task) of that dependency. For example the dependencies illustrated at Figure 2 (Section 1.1) are the dependencies of a software application required for running that application. We can assign to such dependencies a type such as `_run`. Complementarily we can organize dependency types hierarchically to enable 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 we can read it. If D denotes the set of all dependency

types, and types $d,d' \in D$, we shall use $d \triangleright d'$ to denote that d is a subtype of d', e.g. `_edit` ▹ `_read`.

Analogously, and for specializing the very general notion of module to more refined categories, ***module types*** are introduced. For example, the source code of a program in java could be defined as a module of type `SoftwareSourceCode`. Since every source code contains text, `SoftwareSourceCode` can be defined as a subtype of `TextFile`. For this reason module types can be organized hierarchically. If C is the set of all modules and $c,c' \in C$ then $c \triangleright c'$ denotes that c is a subtype of c'.

Since the number of dependency types may increase, it is beneficial to organize them following an object-oriented approach. Specifically we can specify the domain and range of a dependency type to be particular module types. For example the domain of the dependency type `_compile` must be a module denoting source code files while the range must be a compiler. This case is shown in Figure 4 where thick arrows are used to denote subtype relationships.

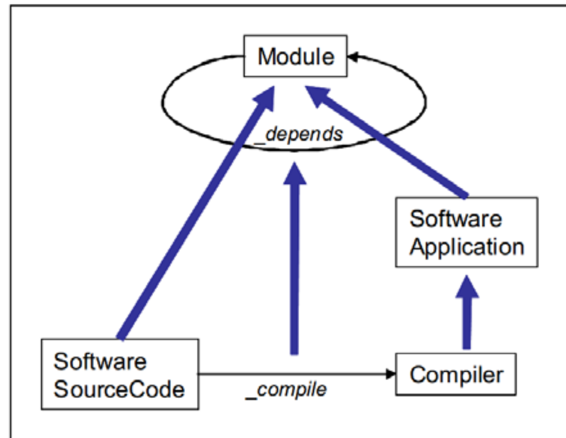

Figure 4: Restricting the domain and range of dependencies

Our model can host dependencies of various interpretations. For instance, the workflow presented at Figure 1 expresses the derivation of data products from other data products and such derivation histories are important for understanding, validating and reproducing the final products. According to our model, data are modeled as modules, while derivation edges are modeled as dependencies (important for the aforementioned tasks). Similarly, our model allows a straightforward modeling of an OAIS representation network (Figure 3), since the notion *interpretedUsing* can be considered as a specialized form of dependency. In addition, a module can have several dependencies of different types. For example, Figure 5 illustrates the dependencies of a file in FITS format where boxes denote modules and arrows denote dependencies of different types (the starting module depends on the ending module, the label denotes the type of the dependency). The file `mars.fits` can be read only with an appropriate application which is modeled with 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 then we must have available the FITS Documentation and the 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), since it allows the definition of dependencies of various interpretations.

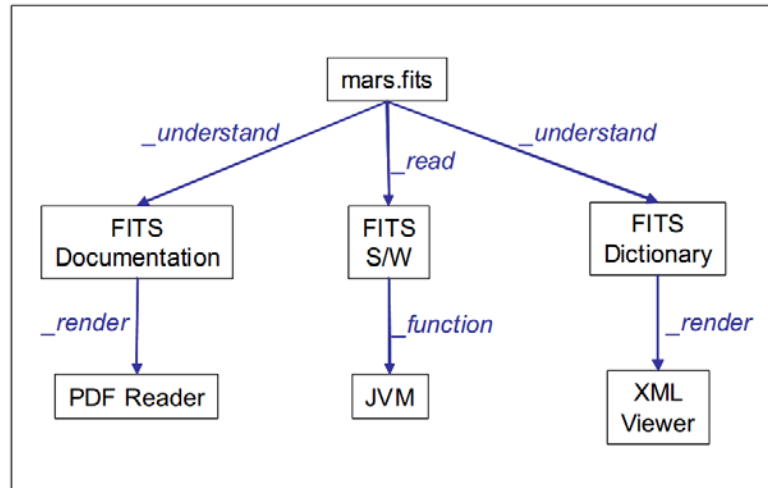

Figure 5: Modeling the dependencies of a FITS file

1.2.1.1 Conjunctive Versus Disjunctive Dependencies

Usually there are more than one ways to perform a task. For example for reading the file `HelloWorld.java` a text editor is required, say `NotePad`. However we can read this file using other text editors, e.g. `VI`. Therefore, and for the task of readability, we could say that `HelloWorld.java` depends on `NotePad` **OR** `VI`. However, the dependencies considered so far are interpreted conjunctively, so it is not possible to capture the above scenario. The disjunctive nature of dependencies was first approached at [10] using the concept of generalized module (which is a set of modules interpreted disjunctively) but the management of such modules was rather complex. A more clear model, based on Horn Rules, which also allows defining the properties (e.g. transitivity) of dependencies straightforwardly, was presented in [21]. Hereafter we will focus on that model. Consider a user's system containing the files `HelloWorld.java`, `HelloWorld.cc` and the software components `javac`, `NotePad`, `VI` and `JVM`, and suppose that we want to preserve the ability to edit, read, compile and run these files. We will model these modules and their dependencies using facts and rules. The digital files are modelled using facts while rules are employed to represent tasks and dependencies. Moreover rules are used for defining module and dependency type hierarchies (i.e. `JavaSourceCode` ⊳ `TextFile`, `_edit` ⊳ `_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 facts describing the digital objects while lines 7-14 are rules denoting various tasks and how they can be carried out. In particular, rules 7 and 8 define an hierarchy of module types (JavaSourceFile ▹ TextFile and C++SourceFile ▹ TextFile), while rules 12-14 define an hierarchy of tasks (Editable ▹ Readable, JavaCompilable ▹ Compilable and C++Compilable ▹ Compilable). Finally, rules 9-11 express which are the tasks that can be performed and which the dependencies of these tasks are (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.

Using the terminology and syntax of Datalog below we define in more detail the modules and dependencies of this example.

Modules – Module Type Hierarchies

Modules are expressed as facts. Since we allow a very general interpretation of what a module can 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:

```
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').
```

Tasks – 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 specifically it defines that every TextFile depends on a TextEditor for editing 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 organize dependency types hierarchically using rules. As already stated, the motivation is to enable deductions of the form "if we can do task A then we can do task B". For example suppose that we want to express that if we can edit a file then certainly we can read it. This can be expressed with the 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`. This kind of inference (transitivity) can be expressed using the rule:

```
Runnable(X,Z) :- Runnable(X,Y) , Runnable(Y,Z).
```

Other properties (i.e. symmetry) that dependencies may have can be defined analogously.

1.2.1.2 Synopsis

Synopsizing we described a method for modeling digital objects and their dependencies.

- 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 typed dependencies between modules.

Therefore the performability of a task determines which the dependencies are. 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` **<u>OR</u>** `VI`).

1.2.2 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 may 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 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 whose Knowledge Base includes the Greek language will be able to understand a Greek text. We can use a similar approach to limit the long chain of dependencies that have to be recorded. A Designated Community is an identified group of users (or systems) that are able to understand a particular set of information, and a DC Knowledge is the information that is assumed to be known from the members of that community. The Knowledge Base of OAIS only makes implicit assumptions about the community knowledge. However in our model we allow making these assumptions *explicit* by assigning to the users of a community the modules that are assumed to be known from them. To this end we introduce the notion of DC Profiles.

Definition 1 If u is a DC (Designated Community), its profile, denoted by $T(u)$, is the set of modules assumed to be known from that community.

As an example, Figure 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`}. This means that 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.

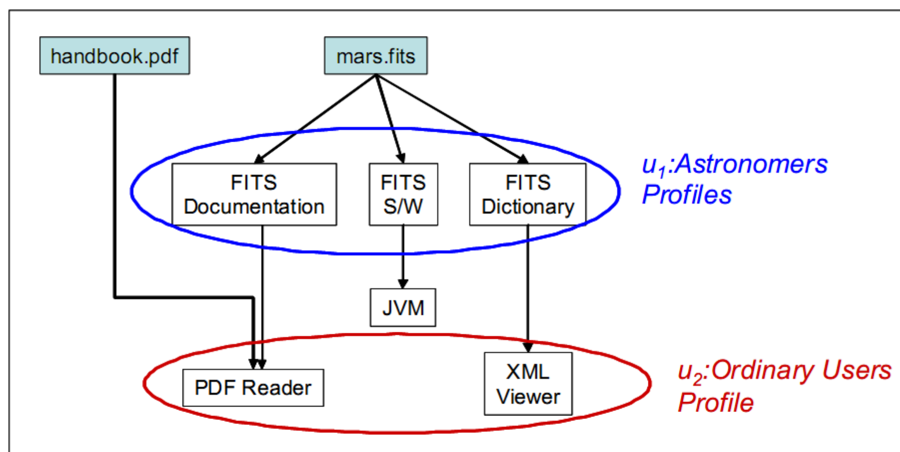

Figure 6: DC Profiles Example

We can assume that the graph formed by the modules and the dependencies is global in the sense that it contains all the modules that have been recorded, and we can assume that the dependencies of a module are always the same

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

Def. 1. implies that if the users of a community know a set of modules they also know how to perform tasks on them. However the performability of tasks is represented with dependencies. 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.

In our example this means that since astronomers know the module `FITS S/W`, 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, 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. 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`.

We will discuss the consequences of not making the assumption expressed in Axiom 1, later in Section 1.2.5.

1.2.3 Intelligibility-related Preservation Services

1.2.3.1 Deciding Intelligibility

Consider two users u_1, u_2 who want to reproduce the music encoded in an mp3 file. The user u_1 successfully reproduces the file while u_2 does not recognize it and requests its direct dependencies. Suppose that the dependency that has been recorded is that the mp3 file depends (regarding reproducibility) on the availability of an mp3-compliant player, say `Winamp`. The user is informed about this dependency and installs the application. However he claims that the file cannot be reproduced. This happens 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. This is an example where the ability to perform a task depends not only on its direct dependencies but also on its indirect dependencies.

This means that to decide the performability of a task on a module by a user, we have to find all the necessary modules by traversing the dependency graph and then to compare them with the modules of the DC profile of the user. However the disjunctive nature of dependencies complicates this task. Disjunctive dependencies express 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 the user. The problem becomes more complicated due to the properties (e.g. transitivity) that dependencies may have (which can be specified using rules as described in Section 1.2.1.1). On the other hand, if all dependencies are conjunctive, then the required modules obtained from the dependency graph are unique.

Below we describe methods for deciding the intelligibility of an object for two settings; the first allows only conjunctive dependencies, while the second permits also disjunctive dependencies.

Conjunctive Dependencies

If dependencies are interpreted conjunctively 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 $Nr^+(t) = \{t' \mid t >^+ t'\}$
- 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. This means that 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 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 the 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.

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 profile u_1) since they already understand all the modules that are required for making mars.fits intelligible, i.e. we have Nr^+(mars.fits) $\subseteq Nr^*(T(u_1))$. However Nr^+(mars.fits) $\not\subset Nr^*(T(u_2))$.

Disjunctive Dependencies

The previous approach will give incorrect results if we have disjunctive dependencies. For example, consider the dependencies shown in Figure 7 and suppose that all of them are disjunctive. This means that o depends on either t_1, or t_2. In this case we have two possible sets for $Nr^+(o)$; $Nr^+(o)=\{t_1, t_3\}$ and $Nr^+(o)=\{t_2, t_4\}$.

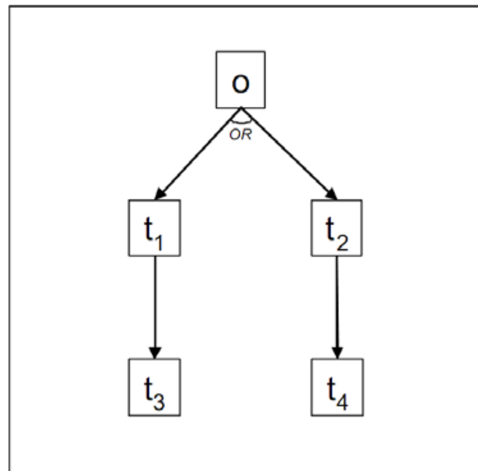

Figure 7: The disjunctive dependencies of a digital object o

For capturing disjunctive dependencies we use the rule-based framework described at Section 1.2.1.1. Figure 8 shows the logical architecture of a system based on facts and rules. The boxes of the upper layer contain information available for all users regarding tasks, dependencies and taxonomies of tasks/ modules. The lower layer contains the modules that are available to each user. Every box corresponds to a user, e.g. *James* has a file HelloWorld.java in his laptop and has installed the applications Notepad and VI. If a user wants to perform a task with a module then he can use the facts of her box and also exploit the rules from the boxes of the upper layer.

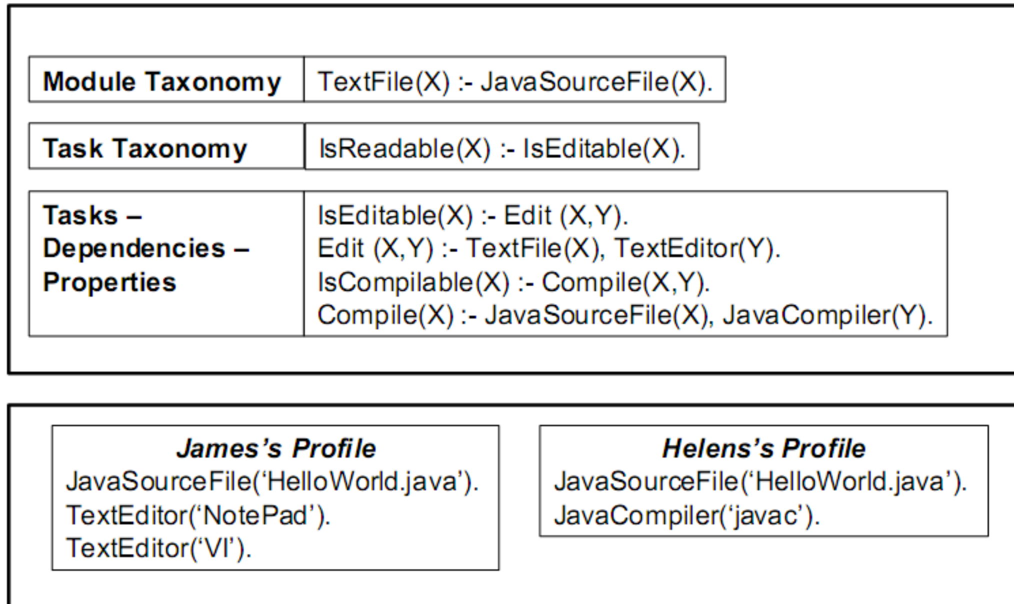

Figure 8: A partitioning of facts and rules

In general, 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` by *James* we can send the Datalog query `IsEditable(X)` using only the facts of his profile. The answer will contain the above module meaning that *James* can successfully perform that task. In fact he can edit it using any of the two different text editors that are available to him. Now suppose that we want to check whether he can compile this file. This requires answering the query `IsCompilable(X)`. 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.

1.2.3.2 Discovering Intelligibility Gaps

If a module is not intelligible from the users of a DC profile, we have an ***Intelligibility Gap*** and we have to provide the extra modules required. In particular, 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 required modules and hence avoiding redundancies. However the disjunctive nature of dependencies makes the computation of intelligibility gap complicated. For this reason we again distinguish the case of conjunctive from the case of disjunctive dependencies.

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. For conjunctive dependencies it holds: $Gap(t,u) = Nr^{+}(t) \setminus Nr^{*}(T(u))$

For example in Figure 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.
Clearly, if a module is already intelligible from a user, the intelligibility gap will be empty i.e. Gap(`mars.fits`,u_1)=$\emptyset$.

Consider the file `HelloWorld.java`, shown in Figure 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. This motivates the need for type-restricted gaps which are described below.

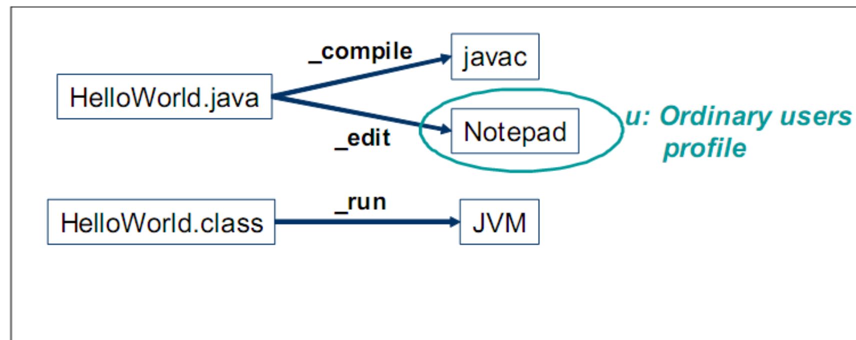

Figure 9: Dependency types and Intelligibility Gap

In general, to compute type-restricted gaps we have to traverse the dependencies of a specific type. However since dependency types are organized hierarchically, we must also include all the subtypes of the given type. Furthermore, 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:

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

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, i.e. $W^* = \bigcup_{d \in W} (\{d\} \cup \{d' | d \rhd d'\})$. This means that $t >_W t'$ holds if $t > t'$ holds and at least one of the types of $t > t'$ belong to W or to a subtype of a type in W. Note that a dependency might be used for the performability of more than one tasks (and therefore it can be associated with more than one types). Now the intelligibility gap between a module t and a user u with respect to a set of dependency types W is defined as:

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

Note that since we want to perform all tasks in W, 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. In our example we have:

Gap(`HelloWorld.java`, *u*, {_edit})= ∅
Gap(`HelloWorld.java`, *u*, {_compile})= {`javac`}
Gap(`HelloWorld.java`, *u*, {_edit, _compile})={`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. 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 8, wants to compile the file `HelloWorld.java`. Since he cannot compile it, he would like to know what he should do. What we have to do in this case is to find and deliver to him the possible facts that would allow him to perform the task.

In order to find the possible explanations of a consequence we can use *abduction* [11][12][13]. *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 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 8). The set M would contain the task that cannot be performed (i.e. the fact `IsCompilable(HelloWorld.java)`). Finally the set H would contain all existing modules that are possible explanations for the performability of the task (i.e. all modules in the lower part of Figure 8). Then `JavaCompiler('javac')` will be an abductive explanation, denoting the adequacy of this module for the compilability of the file. If there are more than one possible explanations (e.g. if there are several java compilers), logic-based abduction would return all of them. However, one can define criteria for picking one explanation as "the best explanation" rather than returning all of them.

1.2.3.3 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 [14] distinguishes packages to AIPs (Archival 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 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 that 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))

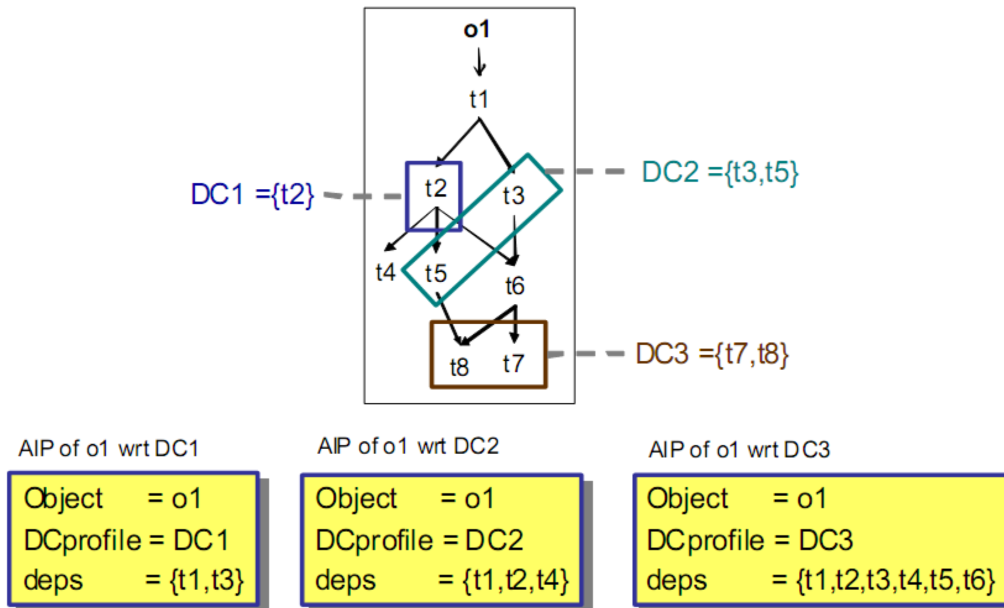

Figure 10: Exploiting DC Profiles for defining the "right" AIPs

Figure 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:

$Pack(o_1, DC_1) = (o_1, \{t_1, t_3)\}$

$Pack(o_1, DC_2) = (o_1, \{t_1, t_2, t_4\})$

$Pack(o_1, DC_3) = (o_1, \{t_1, t_2, t_3, t_4, t_5, t_6\})$

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 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.

We should also note that 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 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.

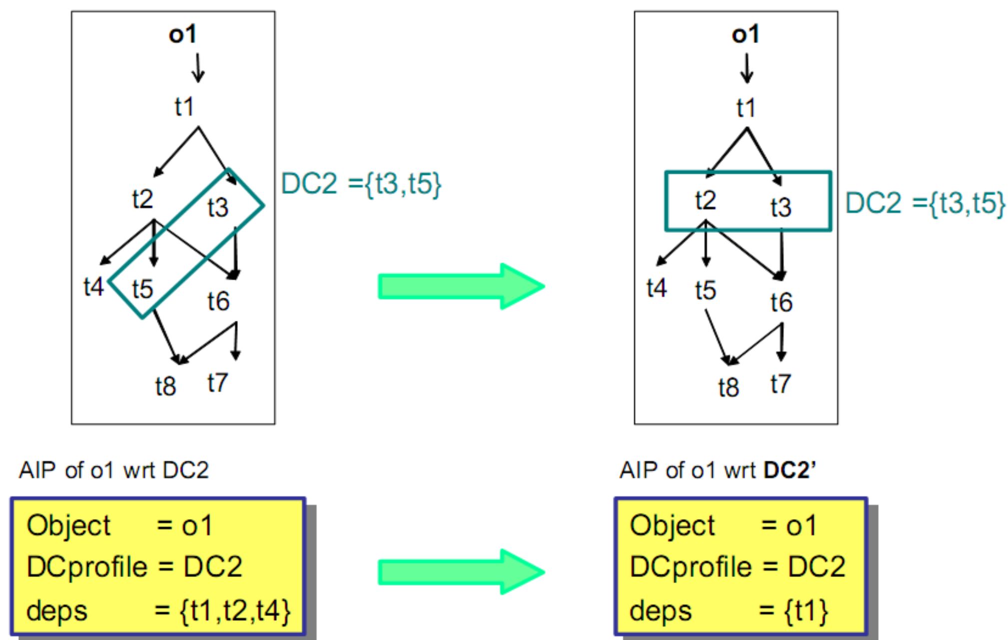

Figure 11: Revising AIPs after DC Profile changes

1.2.3.4 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 (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 A module *t* is *related* with a profile *u*, denoted by $t \mapsto u$, if $Nr^*(t) \cap Nr^*(T(u)) \neq \varnothing$

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 ($RelProf(t) = \varnothing$ in the first and $RelProf(t) \neq \varnothing$ 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, to identify if a module is related with a profile we can compute the intelligibility gap with respect to that profile and with respect to an empty profile (a profile that contains no facts at all). 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 or each case. If the two sets contain the same modules (the same facts) then the module is not be related with that profile. 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.

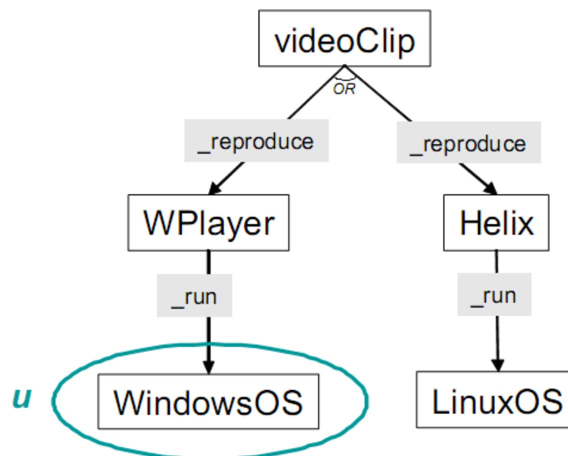

Figure 12: Identifying related profiles when dependencies are disjunctive

As an example, Figure 12 shows the disjunctive dependencies of a digital video file regarding reproduction. 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* respectively. Since there are two ways to reproduce the file, there will be two intelligibility gaps whose unions will contain:

AllGaps1={{WPlayer}, {Helix, LinuxOS}},
so *gap1*={WPlayer, Helix, LinuxOS}
AllGaps1={{WPlayer}, {Helix, LinuxOS}},
so *gap2*={WPlayer, WindowsOS,Helix, LinuxOS}
Since $gap1 \subset gap2$ it follows that videoClip $\mapsto$ *u*.

1.2.4 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 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.

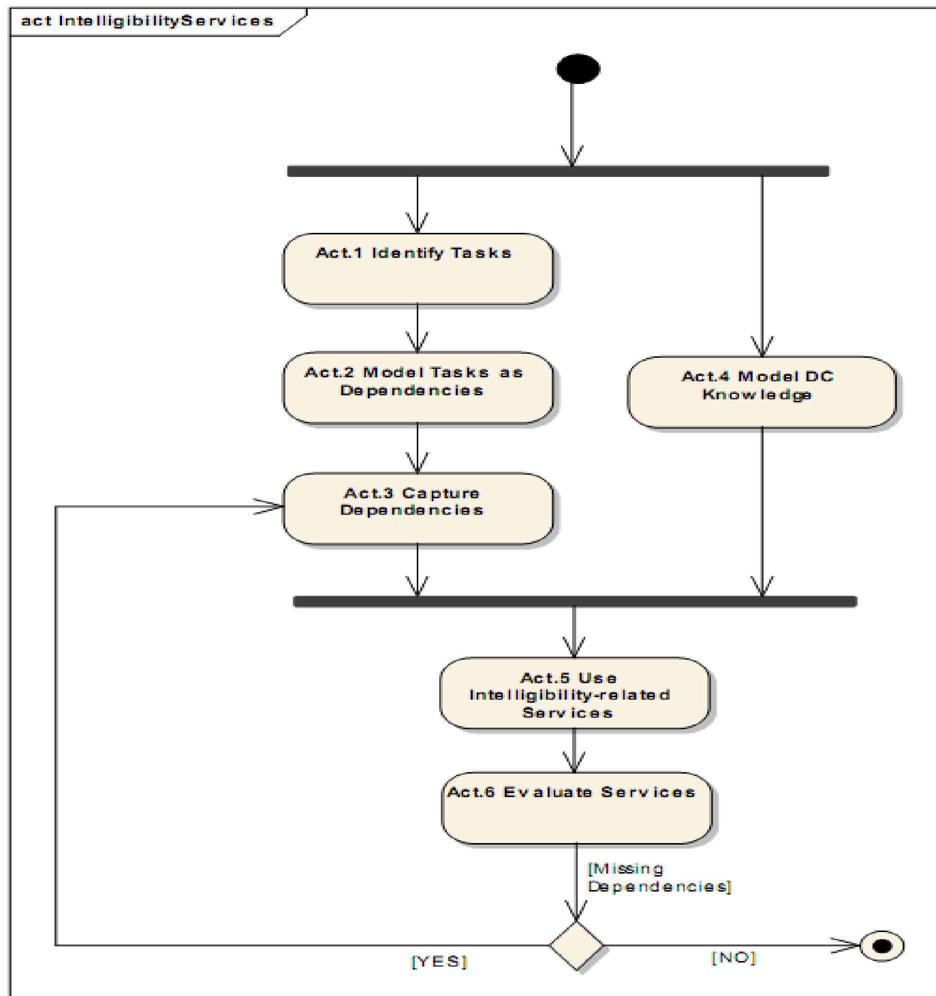

Figure 13: Methodological steps for exploiting intelligibility-related services

Act. 1 **Identification of tasks.** The identification of the tasks whose performance should be preserved with the modules of an archive is very important since these tasks will determine the dependencies types. In our example, the most usual task for digital images is to render them on screen. Since there are two different types of images we can specialize this task to *2Drendering* and *3Drendering*.

Act. 2 **Model tasks as dependencies.** The identified tasks from the previous activity can be modeled using dependency types. If there are tasks that can be organized hierarchically, this should be reflected to the definition of the corresponding dependency types. In our example, we can define the dependency types *render*, *render2D*, and *render3D*, and we can define two subtype relationships, i.e. *render2D* ⊳ *render* and *render3D* ⊳ *render.* We can also define hierarchies of modules e.g. `2DImage` ⊳ `Image`.

Act. 3 **Capture the dependencies of digital objects**. This can be done manually, automatically or semi-automatically. Tools like PreScan[1] 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. In our example we can have dependencies of the form $landscape.jpeg >_{render2D} ImageViewer$, $illusion.3ds >_{remder3D} 3D_Studio$ etc.

Act. 4 **Modeling community knowledge.** This activity enables the selection of those modules that are assumed to be known from the designated communities of users. In our example suppose we want to 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 ordinary users. We should note at this point that Act. 3 and Act. 4 can be performed in any order or in parallel. For example, performing Act. 4 before Act.3 allows reducing the dependencies that have to be captured, e.g. we will not have to analyze the dependencies of `Image Viewer` because that module is already known from both profiles.

Act. 5 **Exploit the intelligibility-related services according to the needs.** We can answer questions of the form: which modules are going to be affected if we remove the `3D Studio` from our system? The answer to this question is the set $\{t \mid t > 3D\ Studio\}$. As another example, we can answer questions of the from: which are the missing modules, if an ordinary user *u* wants to render the 3D image `Illusion.3ds`. In general such services can be exploited for identifying risks, for packaging, and they could be articulated with monitoring and notification services.

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 (unable to perform a task), 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 to him the intelligibility gap which contains only the module `3D Studio`.

[1] Available from: http://www.ics.forth.gr/prescan

However the user claims that the image cannot be rendered. This may happen because there is an additional dependency that has not been recorded, e.g. the fact that the matlib library is required to render a 3D model correctly. 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

1.2.5 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 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 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 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.

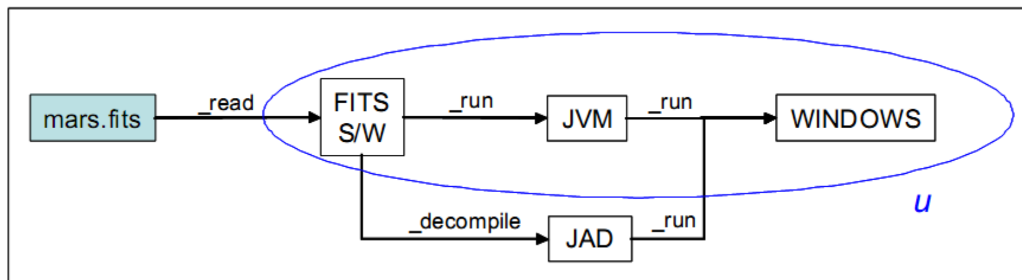

Figure 14: Modeling DC profiles without making any assumptions

This scenario requires changing the definitions of some intelligibility-related because the set of modules understandable by a user u is its profile $T(u)$, and not $Nr^{*}(T(u))$. Intelligibility checking and gap computation are now defined as:

Deciding Intelligibility: True if $Nr^{+}(t) \triangleright T(u)$

Intelligibility Gap: $Nr^{+}(t) \setminus T(u)$

Another consequence is that we cannot reduce the size of DC profiles by 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 the various tasks and their dependencies are expressed explicitly. In this case the performability of a task is modeled using two 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 the modules that are available to the users (i.e. `TextEditor('NotePad').`). 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, 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 at the type layer. Such dependencies can be captured by enriching DC profiles with extensional predicates (again with arity greater than 2) that can express the knowledge of a user to perform a task in a particular way, and associating these predicates with the predicates that concern 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, he could use the following fact in his profile `EditableBy('Readme.txt', 'myProgr.exe').`

1.3 Modeling and Implementation Frameworks

1.3.1.1 Modeling 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 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 (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 information system.

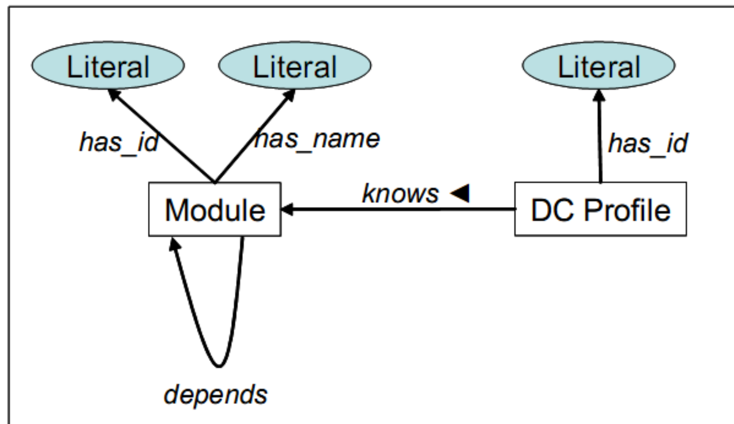

Figure 15: The Core Ontology for representing Dependencies (COD)

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 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 [15] using *wasDerivedFrom* relationships, or according to the CIDOC CRM Ontology [16] using paths of the form:
S22 was derivative created by → **C3 Formal Derivation** → *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 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`") → *S21 was derivation source for* → **C3 Formal Derivation** ("`Compilation`") → *P16 used specific object* → **Module**("`javac`")[2]. 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.

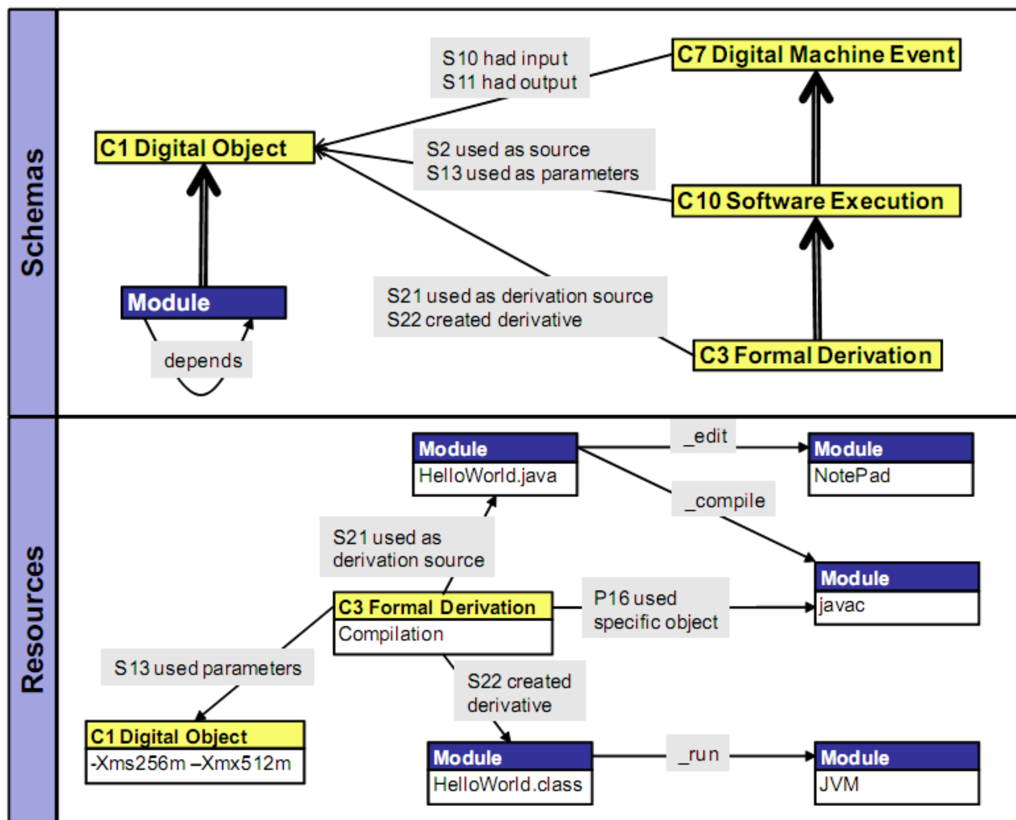

Figure 16: Extending COD for capturing provenance

1.3.2 Implementation Approaches for Disjunctive Dependencies

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

[2] 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 16. These properties have the same interpretation but inverse domain and range.

Prolog is a declarative logic programming language, where a program is a set of Horn clauses describing the data and the relations between them. The proposed approach can be straightforwardly expressed in Prolog. Furthermore, and regarding abduction, there are several approaches that either extend Prolog [17] or augment it [18].

The **Semantic Web Rule Language (SWRL)** [19] is a combination of OWL DL and OWL Lite [20] with the Unary/Binary Datalog RuleML[3]. SWRL provides the ability to write 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. Recursion is required for being able to express the properties (e.g. transitivity) of dependencies. 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 1) synopsizes the various implementation approaches and describes how the elements of the model can be implemented. The table does not contain any information about the Prolog approach since the terminology we used for founding the model (Section 1.2.1.1) is the same with that of Prolog.

What	DB Approach	Semantic Web Approach
Module Type predicates	relational table	class
Facts regarding Modules (and their types)	tuples	class instances
DC Profile	relational table	class
DC Profile 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*
Task Performability	Datalog queries (recursive SQL)	rules

Table 1: Implementation Approaches for Disjunctive Dependencies

[3] http://ruleml.org

References

[1] S.B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. *In Procs of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1345-1350. ACM, 2008.

[2] E. Sunagawa, K. Kozaki, Y. Kitamura, and R. Mizoguchi. An Environment for Distributed Ontology Development Based on Dependency Management. *In Procs of the 2nd International Semantic Web Conference (ISWC'03)*, pages 453-468, 2003.

[3] M.S. Fox and J. Huang. Knowledge provenance: An approach to modeling and maintaining the evolution and validity of knowledge. EIL Technical Report, University of Toronto.

[4] 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 Procs of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, Monterey, California, April 2005.

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

[6] X. Franch and N.A.M. Maiden. Modeling Component Dependencies to Inform their Selection. *In Procs of the 2nd International Conference on COTS-Based Software Systems*, Springer, 2003.

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

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

[9] M. Walter, C. Trinitis, and W. Karl. OpenSESAME: an intuitive dependability modeling environment supporting inter-component dependencies. *In Procs of Pacific Rim International Symposium on Dependable Computing*, pages 76-83, 2001.

[10] Y. Tzitzikas and G. Flouris. Mind the (Intelligibily) Gap. *In Procs of the 11th European Conference on Research and Advanced Technology for Digital Libraries, (ECDL'07)*, Budapest, Hungary, September 2007. Springer-Verlag.

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

[12] L. Console, D.T. Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*. 1(5):661, 1991.

[13] T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM (JACM)*, 42(1):3--42, 1995.

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

[15] L. Moreau, J. Freire, J. Myers, J. Futrelle, and P. Paulson. The Open Provenance Model, *University of Southampton*, 2007.

[16] 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.

[17] H. Christiansen and V. Dahl. Assumptions and abduction in Prolog. *In 3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL*, Saint-Malo, France, September 2004

[18] H. Christiansen and V. Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. *Lecture Notes in Computer Science*, 3668:159-173, 2005.

[19] 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/).

[20] 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, 2002. (http://www.w3c.org/TR/owl-ref).

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