Conversion and Emulation-aware Dependency Reasoning for Curation Services

Yannis Tzitzikas and Yannis Marketakis and Yannis Kargakis
Computer Science Department, University of Crete, Greece
{tzitzik|marketak|kargakis}@ics.forth.gr

ABSTRACT
A quite general view of the digital preservation problem and its associated tasks (e.g. intelligibility and task-performability checking, risk detection, identification of missing resources for performing a task) is to approach it from a dependency management point of view. In this paper we extend past rule-based approaches for dependency management for modeling also converters and emulators and we demonstrate how this modeling allows performing the desired reasoning and thus enables offering more advanced digital preservation services. Specifically these services can greatly reduce the human effort required for periodically checking (monitoring) whether a task on a digital object is performable.

1. INTRODUCTION
In digital preservation there is a need for services that help archivists in checking whether the archived digital artifacts remain intelligible and functional, and in identifying the consequences of probable losses (obsolescence risks). To tackle the aforementioned requirements [14] showed how the needed services can be reduced to dependency management services, and how a semantic registry (compatible with OAIS\(^1\)) can be used for offering a plethora of curation services. Subsequently, [15] extended that model with disjunctive dependencies. The key notions of these works is the notion of module, dependency and profile. In a nutshell, a module can be a software/hardware component or even a knowledge base expressed either formally or informally, explicitly or tacitly, that we want to preserve. A module may require the availability of other modules in order to function, be understood or managed. We can denote such dependency relationships as \( t > t' \) meaning that module \( t \) depends on module \( t' \). A profile is the set of modules that are assumed to be known (available or intelligible) by a user (or community of users), and this notion allows controlling the number of dependencies that have to be recorded formally (or packaged in the context of an encapsulation preservation strategy). Subsequently, and since there is not any objective method to specify exactly which are the dependencies of a particular digital object, [10] extended the model with task-based dependencies where the notion of task is used for determining the dependencies of an object. That work actually introduced an extensible object-oriented modeling of dependency graphs expressed in Semantic Web (SW) languages (RDF/S). Based on that model, a number of services have been defined for checking whether a module is intelligible by a community (or for computing the corresponding intelligibility gap), or for checking the performability of a task. These dependency management services were realized over the available SW query languages. For instance, GapMgr\(^2\) and PreScan\(^3\) [9] are two systems that have been developed based on this model, and have been applied successfully in the context of the EU project CASPAR\(^4\). Subsequently, [16] introduced a rule-based model which also supports task-based dependencies, and (a) simplifies the disjunctive dependencies of [15], and (b) is more expressive and flexible than [10] as it allows expressing the various properties of dependencies (e.g. transitivity, symmetry) straightforwardly. That work actually reduced the problem of dependency management to Datalog-based modeling and query answering.

However, the aforementioned works did not capture converters and emulators. Since conversion (or migration) and emulation are quite important preservation strategies, a dependency management approach should allow modeling explicitly converters and emulators (and analyze them from a dependency point of view, since they have to be preserved too), and exploit them during the offered preservation services. For example, a sequence of conversions can be enough for vanishing an intelligibility gap, or for allowing performing a task. Since there is a plethora of emulation and migration approaches that concern various layers of a computer system (from hardware to software) or various source/target formats (e.g. see [3] for an overview), it is beneficial to use advanced knowledge management techniques for aiding the exploitation of all possibilities that the existing and emerging emulators/converters enable, and assist preservation planning (e.g. [1]). This is crucial since the scale and complexity of information assets and systems evolve towards overwhelming the capability of human archivists and curators (either system administrators, programmers and designers).

\(^1\)Open Archival Information System (ISO 14721:2003).

\(^2\)http://athena.ics.forth.gr/9000/Applications/GapManager/

\(^3\)http://www.ics.forth.gr/isl/PreScan

\(^4\)http://www.casparpreserves.eu/
In a nutshell, the main contributions of this paper are: (a) we extend the rule-based approach of [16] for modeling explicitly converters and emulators, (b) we demonstrate how this modeling apart from capturing the preservability of converters and emulators, enables the desired reasoning regarding intelligibility gaps, task performability, risk detection etc; (c) we introduce an algorithm for visualizing the intelligibility gaps and thus assisting their treatment, and (d) shows how the approach can be implemented using recently emerged Semantic Web tools. The rest of this paper is organized as follows. Section 2 discusses the motivation and the context of our work. Section 3 introduces the rule based modeling and Section 4 discusses the corresponding inference services. Section 5 shows how the approach can be implemented using Semantic Web tools. Finally Section 6 summarizes, discusses related issues and identifies issues for further research.

2. CONTEXT AND BACKGROUND

Migration (according to Wikipedia) is a set of organized tasks designed to achieve the periodic transfer of digital materials from one hardware/software configuration to another, or from one generation of computer technology to a subsequent generation. The purpose of migration is to preserve the integrity of digital objects and to retain the ability for clients to retrieve, display, and otherwise use them in the face of constantly changing technology. Emulation (according to Wikipedia) combines software and hardware to reproduce in all essential characteristics the performance of another computer of a different design, allowing programs or media designed for a particular environment to operate in a different, usually newer environment. Emulation requires the creation of emulators, programs that translate code and instructions from one computing environment so it can be properly executed in another. Popular examples of emulators include QEMU [2], Diocurri [17], etc. There is currently a rising interest on emulators for the needs of digital preservation [8]. Just indicatively, [18] overviews the emulation strategies for digital preservation and discusses related issues, and several recent projects have focused on the development of emulators for the needs of digital preservation (e.g. see [17] and [11]).

In brief, and from a dependency perspective, we could say that the migration process changes the dependencies (e.g. the original digital object depends on an old format, while the migrated digital object now depends on a newer format). Regarding emulation we could say that the emulation process does not change the dependencies of digital objects. An emulator essentially makes available the behavior of an old module (actually by emulating its behavior). It follows that the availability of an emulator can “satisfy” the dependencies of some digital objects, but we should note that the emulator itself has its own dependencies that have to be preserved to ensure its performability. The same also holds for converters.

Running Example

James has a laptop where he has installed the NotePad text editor, the javac 1.6 compiler for compiling Java programs and JRE1.5 for running Java programs (bytecodes). He is learning to program in Java and C++ and to this end, and through NotePad he has created two files, HelloWorld.java and HelloWorld.cc, the first being the source code of a program in java, the second of one in C++. Consider another user, say Helen, who has installed in her laptop the VI editor and JRE1.5.

Suppose that we want to preserve these files, i.e. to ensure that in future James and Helen will be able to edit, compile and run these files. In general, to edit a file we need an editor, to compile a program we need a compiler, and to run the bytecodes of a Java program we need a Java Virtual Machine. To ensure preservation we should be able to express the above.

To this end we could use facts and rules. For example, we could state: A file is editable if it is a TextFile and a TextEditor is available. Since James has two text files (HelloWorld.java, HelloWorld.cc) and a text editor (NotePad), we can conclude that these files are editable by him. By a rule of the form: If a file is Editable then it is Readable too, we can also infer that these two files are also readable. We can define more rules in a similar manner to express more task-based dependencies, such as compilability, runability etc. For our running example we could use the following facts and rules:

<table>
<thead>
<tr>
<th>Facts and Rules</th>
<th>Facts</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>NotePad is a TextEditor</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>VI is a TextEditor</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>HelloWorld.java is a JavaFile</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>HelloWorld.cc is a C++File</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>javac1.6 is a JavaCompiler</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>JRE1.5 is a JVM</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>gcc is a C++Compiler</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rules</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A file is Editable if it is a TextFile and a TextEditor is available</td>
<td>✔</td>
</tr>
<tr>
<td>A file is JavaCompilable if it is a JavaFile and a JavaCompiler is available</td>
<td>✔</td>
</tr>
<tr>
<td>A file is C++Compilable if it is a C++File and a C++Compiler is available</td>
<td>✔</td>
</tr>
<tr>
<td>A file is Compilable if it is JavaCompilable or C++Compilable</td>
<td>✔</td>
</tr>
<tr>
<td>A file is a TextFile if it is JavaFile or C++File</td>
<td>✔</td>
</tr>
<tr>
<td>If a file is Editable then it is Readable</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 1: Modeling the running examples with Facts and Rules

The last two columns indicate which facts are valid for James and which for Helen. From these we can infer that James is able to compile the file HelloWorld.java and that if James sends his TextFiles to Helen then she can only edit them but not compile them since she has no facts about Compilers.

Let us now extend our example with converters and emulators. Suppose James has also an old source file in Pascal PL, say game.pas, and he has found a converter from Pascal to C++, say p2c++. Further suppose that he has just bought a smart phone running Android OS and he has found an emulator of WinOS over Android OS. It should follow that James can run game.pas on his mobile phone (by first converting it in C++, then compiling the outcome, and finally by running over the emulator the executable yielded by the compilation).

Regarding curation services, we have identified the following key requirements

Task-Performability Checking. To perform a task we have to...
perform other subtasks and to fulfill associated requirements for carrying out these tasks. Therefore, we need to be able to decide whether a task can be performed by examining all the necessary subtasks. For example, we might want to ensure that a file is runnable, editable or compilable. This should also exploit the possibilities offered by the availability of converters. For example, the availability of a converter from Pascal to C++, a compiler of C++ over Windows OS and an emulator of Windows OS over Android OS should allow inferring that the particular Pascal file is runnable over Android OS.

Risk Detection. The loss or removal of a software module could also affect the performability of other tasks that depend on it and thus break a chain of task-based dependencies. Therefore, we need to be able to identify which tasks are affected by such removals.

Identification of missing resources to perform a task. When a task cannot be carried out it is desirable to be able to compute the resources that are missing. For example, if Helen wants to compile the file HelloWorld.cc, her system cannot perform this task since there is not any C++ Compiler. Helen should be informed that she should install a compiler for C++ to perform this task.

Support of Task Hierarchies. It is desirable to be able to define task-type hierarchies for gaining flexibility and reducing the number of rules that have to be defined.

Properties of Dependencies. Some dependencies are transitive, some are not. Therefore we should be able to define the properties of each kind of dependency.

Background: Datalog

Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog. As we will model our approach in Datalog this section provides some background material (the reader who is already familiar with Datalog can skip this section).

The basic elements of Datalog are: variables (denoted by a capital letter), constants (numbers or alphanumeric strings), and predicates (alphanumeric strings). A term is either a constant or a variable. A constant is called ground term, and the Herbrand Universe of a Datalog program is the set of constants occurring in it. An atom p(t₁, ..., tₙ) consists of an n-ary predicate symbol p and a list of arguments (t₁, ..., tₙ) such that each tᵢ is a term. A literal is an atom p(t₁, ..., tₙ) or a negated atom ¬p(t₁, ..., tₙ). A clause is a finite list of literals, and a ground clause is a clause which does not contain any variables. Clauses containing only negative literals are called negative clauses, while positive clauses are those with only positive literals in it. A unit clause is a clause with only one literal. HornClauses contain at most one positive literal. There are three possible types of Horn clauses, for which additional restrictions apply in Datalog:

- **Facts** are positive unit clauses, which also have to be ground clauses.
- **Rules** are clauses with exactly one positive literal. The positive literal is called the head, and the list of negative literals is called the body of the rule. In Datalog, rules also must be safe, i.e. all variables occurring in the head also must occur in the body of the rule.

In Datalog, the set of predicates is partitioned into two disjoint sets, EPred and IPred. The elements of EPred denote extensionally defined predicates, i.e. predicates whose extensions are given by the facts of the Datalog programs (i.e. tuples of database tables), while the elements of IPred denote intensionally defined predicates, where the extension is defined by means of the rules of the Datalog program.

In our context, the proposed implementation is described at Section 5.

3. THE RULE-BASED MODEL

In accordance to [16], digital files and profiles (as well as particular software archives or system settings) are represented by facts (i.e. database tuples), while task-based dependencies (and their properties) are represented as Datalog rules. To assist understanding, Figure 1 depicts the basic notions in the form of a rather informal concept map, in the sense that a rule-based approach cannot be illustrated with a graph in a manner both intuitive and precise.

![Figure 1: Informal concept map](image)

**Digital Files, Type Hierarchies, and Profiles**

Digital files and their types are represented as EDB facts using predicates that denote their types, e.g. for the three files of our running example we can have the facts shown in the left column of the following table. Software components are described analogously (e.g. see right column).

<table>
<thead>
<tr>
<th>Facts for digital files</th>
<th>Facts for software components</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaFile(HelloWorld.java).</td>
<td>TextEditor(vi).</td>
</tr>
<tr>
<td>C++File(HelloWorld.cc).</td>
<td>JVM(jre1.5win)</td>
</tr>
<tr>
<td>PascalFile(game.pas).</td>
<td>JVM(jre1.6linux)</td>
</tr>
</tbody>
</table>

Each file can be associated with more than one type. In general we could capture several features of the files (apart from types) using predicates (not necessarily unary), e.g. LastModifDate(HelloWorld.java, 2008-10-18).

The types of the digital files can be organized hierarchically, and such taxonomies can be represented with rules, e.g. to define that every JavaFile is also a UTF8File we must add the rule UTF8File(X) :- JavaFile(X).
A profile is a set of facts, describing the modules available (or assumed to be known) to a user (or community). For example, the profiles of James and Helen are the ticked facts in the corresponding columns of Table 1.

**Task Dependencies and Task Hierarchies**

We will also use (P,x) predicates to model tasks and their dependencies. Specifically, for each real world task we define two intensional predicates: one (which is usually unary) to denote the (performability of the) task, and another one (with arity greater than one) for denoting the dependencies of the task. For instance, Compile(HelloWorld.java) will denote the compilability of HelloWorld.java. Since its compilability depends on the availability of a compiler (specifically a compiler for the Java language), we can express this dependency using a rule of the form: Compile(X) :- Compilable(X, Y) where the binary predicate Compilable(X, Y) is used for expressing the appropriateness of a Y for compiling a X. For example, Compilable(HelloWorld.java, javac 1.6) expresses that HelloWorld.java is compilable by javac 1.6. It is beneficial to express such relationships at the class level (not at the level of individuals), specifically over the types (and other properties) of the digital objects and software components, i.e. with rules of the form:

- Compilable(X, Y) :- JavaFile(X), JavaCompiler(Y).
- Compilable(X, Y) :- C++File(X), C++Compiler(Y).
- Runable(X, Y) :- JavaFile(X), JavaCompiler(Y).
- Editable(X, Y) :- JavaFile(X), TextEditor(Y).

Relations of higher arity can be employed based on the requirements, e.g.:

- Run(X) :- Runable(X, Y, Z)
- Runable(X, Y, Z) :- JavaFile(X), Compilable(X, Y), JVM(Z)

We can express hierarchies of tasks as we did for file type hierarchies, for enabling deductions of the form: “if we can do task A then certainly we can do task B”, e.g. “if we can edit something then certainly we can read it too” expressed as: Read(X) :- Edit(X).

We can also express general properties of task dependencies, like transitivity. For example, from Runable(a.class, JVM) and Runable(JVM, Windows) we might want to infer that Runable(a.class, Windows). Such inferences can be specified by a rule of the form:

- Runable(X, Y) :- Runable(X, Z), Runable(Z, Y).

As another example, IntelligibleBy(X, Y, Z) :- IntelligibleBy(X, Z), IntelligibleBy(Z, Y). This means that if X is intelligible by Z and Z is intelligible by Y, then X is intelligible by Y. This captures the assumptions of the dependency model described in [14] (i.e. the transitivity of dependencies).

**Modeling Converters**

Conversions are special kinds of tasks and are modeled differently. In brief to model a converter and a corresponding conversion we have to introduce one unary predicate for modeling the converter (as we did for the types of digital files) and one rule for each conversion that is possible with that converter (specifically one for each supported type-to-type conversion).

In our running example, consider the file game.pas (which contains source code in Pascal PL) and the converter p2c++ from Pascal to C++. Recall that James has a compiler for C++. It follows that James can compile game.pas since he can first convert it in C++ (using the converter), then compile it and finally run it. To capture the above scenario it is enough to introduce a predicate for modeling the converters from Pascal to C++, say ConverterPascal2C++, and adding the following rule:

- C++File(X) :- PascalFile(X), ConverterPascal2C++(Y).

Since the profile of James will contain the facts PascalFile(game.pas) and ConverterPascal2C++(p2c++), we will infer C++File(game.pas), and subsequently that this file is compilable and runnable.

Finally we should not forget that a converter is itself a module with its own dependencies, and for performing the intended task the converter has to be runnable. Therefore, we have to update the rule as follows:

- C++File(X) :- PascalFile(X), ConverterPascal2C++(Y), Run(Y).

**Modeling Emulators**

Emulation is again a special kind of task and is modeled differently. Essentially we want to express the following: (i) If we have a module X which is runnable over Y, (ii) and an emulator E of Y over Z (hosting system=Z, target system=Y), (iii) and we have Z and E, (iv) then X is runnable over Z. For example, consider the case where:

- X=a.exe (a file which is executable in Windows operating system),
- Y=WinOS (the Windows operating system),
- Z=AndroidOS (the Android operating system), and
- E=W4A (i.e. an emulator of WinOS over AndroidOS).

In brief, for each available emulator (between a pair of systems) we can introduce a unary predicate for modeling the emulator (as we did for the types of digital files, as well as for the converters), and writing one rule for the emulation.

For example, suppose we have a file named a.exe which is executable over WinOS. For this case we would have written:

- Run(X) :- Runable(X, Y)
- Runable(X, Y) := WinExecutable(X), WinOS(Y)

and the profile of a user that has this file and runs WinOS would contain the facts WinExecutable(a.exe) and WinOS(mycomputer), and by putting them together it follows that Run(a.exe) holds. Now consider a different user who has the file a.exe but runs AndroidOS. However suppose that he has the emulator W4A (i.e. an emulator of WinOS over AndroidOS). The profile of that user would contain:

- WinExecutable(a.exe)
- AndroidOS(mycomputer) // instead of WinOS(mycomputer)
- EmulatorWinAndroid(W4A)

To achieve our goal (i.e. to infer that a.exe is runnable), we have to add one rule for the emulation. We can follow two approaches. The first is to write a rule that concerns the runable predicate, while the second is to write a rule for classifying the system that is equipped with the emulator to the type of the emulated system:

A. Additional rule for Runable

This relies on adding the following rule:

- Runable(X, Y) := Runable(X, Z), Runable(Z, Y), WinOS(Z)

For example, suppose we have a file named a.exe which is executable over WinOS. For this case we would have written:

- Run(X) := Runable(X, Y)
- Runable(X, Y) := WinExecutable(X), WinOS(Y)

and the profile of a user that has this file and runs WinOS would contain the facts WinExecutable(a.exe) and WinOS(mycomputer), and by putting them together it follows that Run(a.exe) holds. Now consider a different user who has the file a.exe but runs AndroidOS. However suppose that he has the emulator W4A (i.e. an emulator of WinOS over AndroidOS). The profile of that user would contain:

- WinExecutable(a.exe)
- AndroidOS(mycomputer) // instead of WinOS(mycomputer)
- EmulatorWinAndroid(W4A)

To achieve our goal (i.e. to infer that a.exe is runnable), we have to add one rule for the emulation. We can follow two approaches. The first is to write a rule that concerns the runable predicate, while the second is to write a rule for classifying the system that is equipped with the emulator to the type of the emulated system:

A. Additional rule for Runable

This relies on adding the following rule:

- Runable(X, Y) := Runable(X, Z), Runable(Z, Y), WinOS(Z)
Runnable(X,Y,Z):- WinExecutable(X),
             EmulatorWinAndroid(Y), AndroidOS(Z)

Note that since the profile of the user contains the fact
EmulatorWinAndroid(W4A) the body of the rule is satisfied
(for X=a.exe, Y=W4A, Z=myComputer), i.e. the rule will yield
the desired inferred tuple Runnable(a.exe,W4A,myComputer).

Note that here we added a rule for the runnable which has
3 variables signifying the ternary relationship between exe-
cutable, emulator and hosting environment.

B. Additional type rule (w.r.t. the emulated Behavior)
An alternative modeling approach is to consider that if a
system is equipped with one emulator then it can also op-
cerate as the emulated system. In our example this can be
expressed by the following rule:
WinOS(X):- AndroidOS(X), EmulatorWinAndroid(Y).

It follows that if the system has an emulator of
type EmulatorWinAndroid (here W4A) and myComputer is of
type AndroidOS, then that rule will infer that WinOS(myComputer),
implying that the file a.exe will be inferred to be runnable
due to the basic rule of runnable which is independent of
emulators (i.e. due to the rule
Runnable(X,Y) :- WinExecutable(X), WinOS(Y)).

Both (A and B) approaches require the introduction of a new
unary predicate about the corresponding pair of systems, here
EmulatorWinAndroid. Approach (A) requires intro-
ducing a rule for making the predicate runnable “emulator-
aware”, while approach (B) requires a rule for classifying
the system to the type of the emulated system. Since em-
umators are modules that can have their own dependencies,
they should be runnable in the hosting system. To require
their runnability during an emulation we have to update the
above rules as follows (notice that last atom in the bodies
of the rules):

A’: Runnable(X,Y,Z):-  B’: WinOS(X):-
               WinExecutable(X),               AndroidOS(X),
               EmulatorWinAndroid(Y),         EmulatorWinAndroid(Y),
               AndroidOS(Z),                Runnable(Y,X)
               Runnable(Y,Z)                Runnable(Y,Z)

Synopsis To synopsize, methodologically for each real world
task we define two intentioned predicates: one (which is usu-
ally unary) to denote the performability of the task, and
another one (which is usually binary) for denoting the de-
pendencies of task (e.g. Read and Readable). To model a
converter and a corresponding conversion we have to intro-
duce one unary predicate for modeling the converter (as we
did for the types of digital files) and one rule for each con-
version that is possible with that converter (specifically one
for each supported type-to-type conversion). To model an
emulator (between a pair of systems) we introduce a unary
predicate for modeling the emulator and writing one rule for
the emulation. Regarding the latter we can either write a
rule that concerns the runnable predicate, or write a rule
for classifying the system that is equipped with the emu-
lator to the type of the emulated system. Finally, and since
converters and emulators are themselves modules, they have
their own dependencies, and thus their performability and
dependencies (actually their runnability) should be modeled
too (as in ordinary tasks).

4. REASONING SERVICES
In general, Datalog query answering and methods of logi-
cal inference (i.e. deductive and abductive reasoning) are
exploited for enabling the required inference services (per-
formability, risk detection, etc.). Here we describe how the
reasoning services described at Section 2 can be realized in
the proposed framework.

Task-Performability. This service aims at answering if a
task can be performed by a user/system. It relies on query
answering over the Profiles of the user. E.g. to check if
HelloWorld.cc is compilable we have to check if HelloWorld.cc
is in the answer of the query Compile(X). As we described
earlier, converters and emulators will be taken into account,
meaning that a positive answer may be based on a complex
sequence of conversions and emulations. This is the essential
benefit from the proposed modeling. Furthermore, classical
automated planning, e.g. the STRIPS planning method [6],
could be applied for returning one of the possible ways to
achieve (perform) a task. This is useful in case there are
several ways to achieve the task.

Risk-Detection. Suppose that we want to identify the conse-
quences on editability after removing a module, say NotePad.
To do so: (a) we compute the answer of the query Edit(X),
let A be the returned set of elements, (b) we delete NotePad
from the database and we do the same, let B be the re-
turned set of elements5, and (c) we compute and return the
elements in A \ B (they are the ones that will be affected).

Computation of Gaps (Missing Modules). The gap is actu-
ally the set of facts that are missing and are needed to per-
form a task. There can be more than one way to fill a gap
due to the disjunctive nature of dependencies since the same
predicate can be the head of more than one rules (e.g. the
predicate TextEditor in the example earlier). One method
for informing the curator about the possible ways to fill it is
to construct and visualize a graph that contains information
about only the related facts and rules. We propose a graph
which is actually a form of AND-OR graph. The user can spec-
dify the desired depth of that graph, or interactively de-
cide to increase the depth gradually. The graph is actually
a compact method for presenting the (possibly numerous)
ways to fill a gap. The construction of the graph resem-
bles the way planning algorithms (in particular backwards
search-based planners) operate. The algorithm starts from
the goal and shows the corresponding rules for achieving that
goal. Those atoms of the rules which have a grounding that
belong to (or can be inferred from) the facts of the profile
at hand, are visualized differently (e.g. colored in green, or
enclosed in squares) so that the user can discriminate the
missing from the available facts. Figure 2 shows some in-
dicative examples. In all cases the goal is a grounded atom,
i.e. a(1), however the rules and the recorded facts are dif-
f erent in each case. In case (I) the graph shows that the gap
is a grounded atom (i.e. a(1)), while in case (II) the graph
shows that the gap is a non grounded atom (i.e. a(var)).
Case (III) demonstrates a case where more than one rules
with the same head are involved, and the depth of the graph
is greater than one. The graph makes evident that there are
two possible ways to fill the gap; according to the first the

5In an implementation over Prolog, we could use the retract
feature to delete a fact from the database.

Page 42
The complexity of the algorithm is in $O^*$ (i.e., the cost of query answering), then the time and $Q$ gap comprises two non grounded atoms (i.e., $D(var)$ and $E(var)$), while according to the second it consists of one non grounded atom (i.e., $D(var)$).

A recursive algorithm for producing such graphs is given (in pseudocode) at Figure 3. The algorithm takes as input a goal (an atom grounded or not), a depth (a positive integer $\geq 1$) and a prevNode (the previous node, it is used only for the recursive calls). Initially, the algorithm is called with the goal of the user (which is a grounded atom) plus the desired depth, and an empty (null) prevNode. The algorithm constructs and returns the corresponding tree graph (like those of Figure 2), whose layout can be derived by adopting one of the several hierarchical graph drawing algorithms.

Figure 2: Three examples of gap graphs

Figure 4 shows a small example of a graph of depth equal to 2 where conversion is involved. The graph corresponds to a case where a file $a.pas$ is compilable. The graph makes evident that to turn $a.pas$ compilable either a PascalCompiler is required or a runnable Pascal2Java converter. Note that if we had a greater depth, then the expansion of Pascal2Java(var1) and Run(var), would not necessarily use the same grounding for var1 and var2, although that would be desired. This and other ways to “inject reasoning” in the graph construction is a subject for further research.

Note that the algorithm returns always a tree and it does not do any arbitrary grounding; it is only the original grounded atom (i.e. the original goal) that is propagated based on the rules. Of course if there are rules whose body contain grounded atoms, the latter appear as such in the graph. The algorithm also does not expand a ground atom if inferred.

Complexity. If $|R|$ denotes the number of rules, $d$ the depth, and $Q$ denotes the cost to check whether a fact exists or is inferred (i.e. the cost of query answering), then the time complexity of the algorithm is in $O(d*Q*|R|)$. Since $|R|$ is usually low, $d$ is an input parameter which again cannot be very big, we can say that the complexity is low.

5. IMPLEMENTATION

There are several possible implementation approaches. Below we describe one Semantic Web-based implementation using RDF/S and OpenLink Virtuoso which is a general purpose RDF triple store with extensive SPARQL and RDF support [5]. Its internal storage method is relational, i.e., RDF triples are stored in tables in the form of quads $(g, s, p, o)$ where $g$ represents the graph, $s$ the subject, $p$ the predicate and $o$ the object. We decided to use this system be-

Algorithm GapGraph (goal: Atom, depth: Integer, prevNode: Node): Node
(01) If (prevNode=null) then
(02) $gNode = Create node(goal)$
(03) else
(04) $gNode = prevNode$
(05) $hrs = all rules having the predicate of the goal as head$
(06) If (|hrs| = 0) then {} // the goal predicate is not head in any rule
(07) headNode = $gNode$
(08) return headNode
(09) end
(10) For each $hr$ in $hrs$
(11) If (|$hrs$| > 1) then {} // there are > 1 rules having the same head
(12) ORnode = create node(ORnode)
(13) create link($gNode$→ORnode)
(14) headNode = ORnode
(15) end
(16) headNode = $gNode$
(17) If (IsGrounded(goal)) then {} // e.g. consider the goal A(1)
(18) Ground the corresponding variable in all atoms of the
(19) body of the rule hr that contain that variable
(20) end
(21) Let BodyAtoms be the resulting set of body atoms
(22) // if the previous step did not ground anything,
(23) then BodyAtoms contains the original body atoms
(24) for each atom in BodyAtoms
(25) atomNode = Create node(atom)$
(26) Create link(headNode→atomNode)
(27) If (IsGrounded(atom)) and
(exists in the fact set (or it can be inferred)) then
(28) Square(AtomNode)
(29) end
(30) If (depth > 1) then
(31) For each atom in BodyAtoms
(32) If (Square(AtomNode)=False) then {
//atomNode corresponds to atom
(33) newNode = GapGraph(atom, depth − 1, atomNode)
(34) Create link(atomNode→newNode)
(35) }
(36) }
(37) }
(38) Return headNode

Figure 3: The algorithm that produces gap graphs

Figure 4: A visualization of a gap graph that involves a converter
cause of its inference capabilities, namely backward chaining reasoning, meaning that it does not materialize all inferred facts, but computes them at query level. Its reasoner covers the related entailment rules of rdfs:subClassOf and rdfs:subPropertyOf, while user defined custom inference rules can be expressed using rule sets. Practically this means that transitive relations (i.e. subClassof, subPropertyOf, etc.) are not physically stored in the knowledge base, but they are added to the result set at query answering. Transitivity is also supported in two different ways. Given a RDF schema and a rule associated with that schema, the predicates rdfs:subClassOf and rdfs:subPropertyOf are recognized and the inferred triples are derived when needed. In case of another predicate, the option for transitivity has to be declared in the query.

For our case, we have to “translate” our facts and rules to quads of the form (g, s, p, o) which are actually RDF triples contained in a graph g. The support of different graphs is very useful for the cases of profiles; we can use a different graph for each profile. We will start by showing how facts can be “translated” to RDF quads and later we will show how inference rules can be expressed using ASK and CONSTRUCT or INSERT SPARQL queries. Note that if we use INSERT instead of CONSTRUCT then the new inferred triples will be stored in the triple store (materialization of inferred triples). Hereafter we will use only CONSTRUCT. For better readability of the SPARQL statements below we omit namespace declarations.

**Modules:** Module types are modeled using RDF classes while the actual modules are instances of these classes. Module type hierarchies can be defined using the rdfs:subClassOf relation. For example the fact JavaFile('HelloWorld.java') and the rule for defining the module type hierarchy TextFile(X) :- JavaFile(X) will be expressed using the following quads:

```
g, <JavaFile>, rdf:type, rdfs:Class
```

**Profiles:** We exploit the availability of graphs to model different profiles, e.g. we can model the profiles of James and Helen (including only some indicative modules), as follows:

```
<jGrph>, <NotePad>, rdf:type, <TextEditor>
<jGrph>, <HelloWorld.java>, rdf:type, <JavaFile>
<jGrph>, <javac_1_6>, rdf:type, <JavaCompiler>
<hGrph>, <VI>, rdf:type, <TextEditor>
<hGrph>, <jre_1_5>, rdf:type, <JavaVirtualMachine>
```

**Dependencies:** The rules governing the performance of tasks and their dependencies are transformed to appropriate SPARQL CONSTRUCT statements which produce the required inferred triples. For example, the rule about the compilability of Java files

(Compilable(X,Y) :- JavaFile(X),JavaCompiler(Y)) is expressed as:

```
CONSTRUCT{?x <compilable> ?y}
WHERE{?x rdf:type <JavaFile>. 
     ?y rdf:type <JavaCompiler>}
```

Finally the unary predicate for the performability of task, here Compile, is expressed as:

```
CONSTRUCT{?x rdf:type <Compile>}
WHERE{ {?x <compilable> ?y} }
```

**Converters:** The rules regarding conversion are modeled analogously, e.g. for the case of a converter from Pascal to C++ we produce:

```
CONSTRUCT{?x rdf:type <C++File>}
WHERE{ {?x rdf:type <PascalFile>. 
           ?y rdf:type <ConverterPascal2C++>. 
           ?y rdf:type <Run>} }
```

**Services:** To realize the reasoning services (e.g. task performance, risk detection, etc), we rely on SPARQL queries. For example to answer if the file HelloWorld.java can be compiled we can send the INSERT query about the compilability of the file (as shown previously) and then perform the following ASK query on the entailed triples:

```
ASK<HelloWorld.java> <compilable> ?y)
```

If this query returns true then there is at least one appropriate module for compiling the file.

The risk-detection service requires SELECT and DELETE SPARQL queries (as discussed at section 4). For example to find those modules whose editability will be affected if we remove the module Notepad, we have to perform

```
SELECT ?x
WHERE {?x rdf:type <Edit>}
```

DELETE <Notepad> rdf:type <TextEditor>

From the select query we get a set A containing all modules which are editable. Then we remove the triple about
Notepad and perform again the select query, getting a new set B. The set difference A \ B will reveal the modules that will be affected. If empty this means that there will be no risk in deleting the Notepad.

Based on the above approach we have implemented a prototype system. Its repository containing the facts and rules of the examples of this paper, and behaving as specified by the theory is accessible through a SPARQL endpoint http://139.91.183.78:8890/sparql.

6. CONCLUDING REMARKS
In this paper we have extended past rule-based approaches for dependency management for capturing converters and emulators, and we have demonstrated that the proposed modeling enables the desired reasoning regarding task performability, which in turn can greatly reduce the human effort required for periodically checking or monitoring whether a task on an archived digital object is performable.

We should clarify that we do not focus on modeling, logging or reasoning over composite tasks in general (as for example it is done in [4]). We focus on the requirements for ensuring the performability of simple (even atomic) tasks, since this is more aligned with the objectives of long term digital preservation. Neither we focus on modeling or logging the particular workflows or derivation chains of the digital artifacts, e.g. using provenance models like OPM or CRM Dig [13]. We focus only the dependencies for carrying out the desired tasks. Obviously this view is less space consuming, e.g. in our running example we do not have to record the particular compiler that was used for the derivation of an executable (and its compilation time), we just care to have at our disposal an appropriate compiler for future use. However, if a detailed model of the process is available, then the dependency model can be considered as a read-only view of that model.

As regards applicability, note that some tasks and their dependencies can be extracted automatically as it has been demonstrated in [9, 7]. As regards available datasets, [12] describes the P2 registry, which uses Semantic Web technologies to combine the content of the PRONOM Technical Registry, represented as RDF, with additional facts from DBpedia, currently containing about 44,000 RDF statements about file formats and preservation tools.

In the near future we plan to further elaborate on gap visualization methods, while issues for future research include composite objects (e.g. software components, systems), update requirements, and quality-aware reasoning for enabling quality-aware preservation planning.

Acknowledgements
Work done in the context of NoE APARSEN (Alliance Permanent Access to the Records of Science in Europe, FP7, Proj. No 269977), and SCIDIP-ES (SCIence Data Infrastructure for Preservation - Earth Science, FP7).

7. REFERENCES