Noname manuscript No.
(will be inserted by the editor)

Assisting Digital Interoperability and Preservation through Advanced Dependency Reasoning

Yannis Tzitzikas · Yannis Kargakis · Yannis Marketakis

the date of receipt and acceptance should be inserted later

Abstract Digital material has to be preserved not only against loss or corruption, but also against changes in its ecosystem. A quite general view of the digital preservation problem is to approach it from a *dependency management* point of view. In this paper we present a *rule*-based approach for dependency management which can model also *converters* and *emulators*. We show that this modeling approach enables the automatic reasoning needed for reducing the human effort required for checking (and monitoring) whether a task on a digital object is performable. We provide examples demonstrating how real world *converters* and *emulators* can be modeled, and show how the preservation services can be implemented. Subsequently, we detail an implementation based on Semantic Web technologies, we describe the prototype system `Epimenides` which demonstrates the feasibility of the approach, and finally we report various promising evaluation results.

1 Introduction

Digital material has to be preserved not only against loss or corruption, but also against hardware/software technology changes, plus changes in the knowledge of the community. 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). Past works [28], [29] and [31] have shown how the aforementioned services can be approached from a *dependency management* perspective. However, past works did not capture *converters* and *emulators*. Since conversion (or migration) and emulation are very fundamental 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. This is of paramount importance since a sequence of conversions and emulations can be enough for vanishing a gap that prevents performing a task. Note that there is a plethora of emulation and migration approaches and tools that concern various layers of a computer system (from hardware to software), or various source/target formats (e.g. see [10] for an overview). This means that 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. [3]). 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).

Below we attempt to pass the main message through an example. Consider a user, say Yannis, who would like to run on his mobile phone, software source code written before many years, e.g. software code written in Pascal programming language, stored in a file named `game.pas`. For example consider the situation illustrated in Figure 1a. The rising questions are:

- What can Yannis do? (to achieve his objective)
- What should we (as community) do?
  - Do we have to develop a Pascal compiler for Android OS?

Yannis Tzitzikas · Yannis Kargakis · Yannis Marketakis
Institute of Computer Science, FORTH-ICS, GREECE
E-mail: {tzitzik, kargakis, marketak}@ics.forth.gr

Yannis Tzitzikas · Yannis Kargakis
Computer Science Department, University of Crete, GREECE
E-mail: {tzitzik, kargakis}@csd.uoc.gr

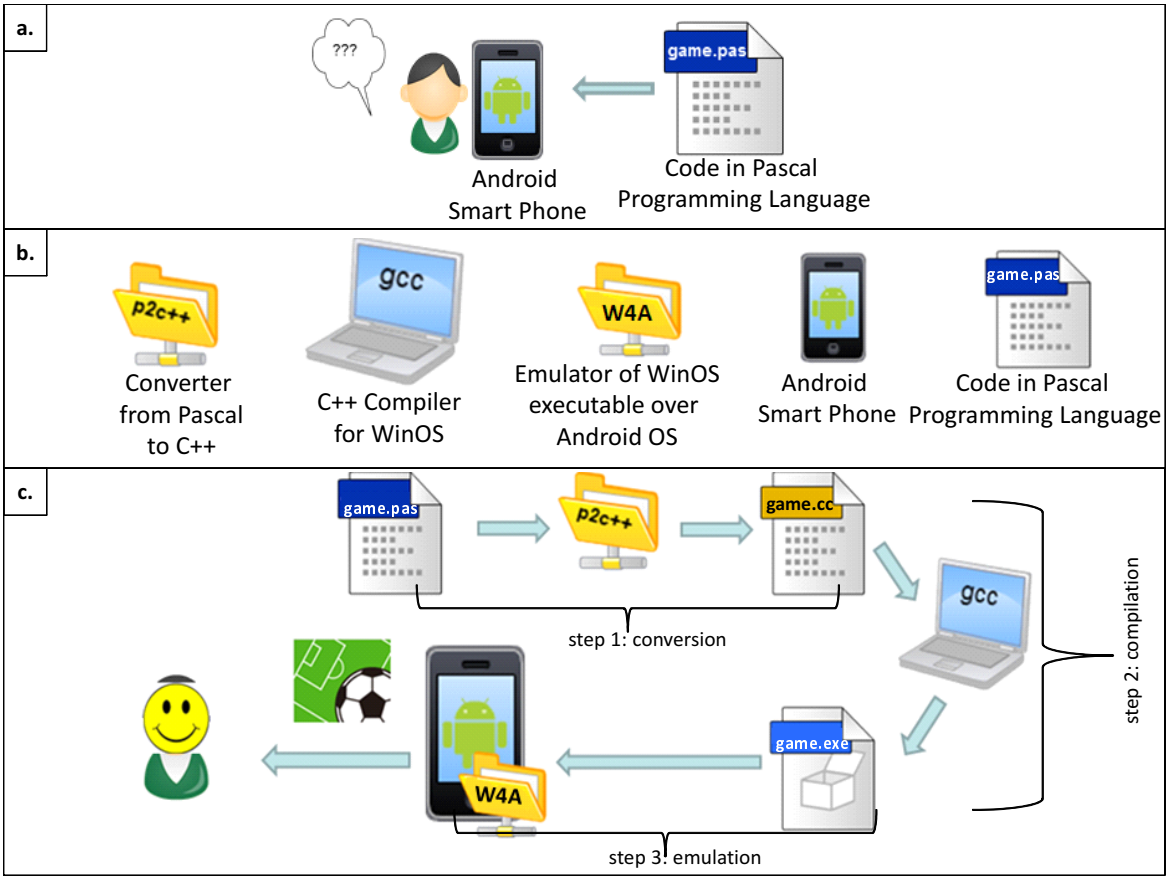

Fig. 1 Running example. (a) The situation, (b) The available modules, (c) A series of conversion/emulation to achieve our objective

- Do we have to standardize programming languages?
- Do we have to standardize operating systems, virtual machines, and so on?

The direction and answer (according to this paper), is that it is worth investigating whether it is already possible to run that code on android by "combining" existing software, i.e. by applying a series of transformations and emulations.

To continue this example, suppose that we have in our disposal only the following (as shown in Figure 1b):

- a converter from Pascal source code to C++ source code (say `p2c++`),
- a C++ compiler (`gcc`) for Windows OS,
- an emulator of Windows OS executable over Android OS (say `W4A`).
- a smart phone running Android OS
- a Pascal File (`game.pas`)

It seems that we could run `game.pas` on his mobile phone in three steps: (step 1) by first converting the Pascal code to C++ code, (step 2) then compiling the C++ code to produce executable code, and finally (step 3) by running over the emulator the executable yielded by the compilation. Indeed, the series of transformation/emulations shown in Figure 1c could achieve our objective.

One might argue that this is very complex for humans. Indeed this is true, and this is why we believe that such reasoning should be done by computers, not humans. The work that we present in the current paper shows how we can model our information in a way that enables this kind of *automated reasoning*.

Although the above scenario concerns *software*, we should clarify that the proposed approach is not confined to software. Various interoperability objectives that concern *documents* and *datasets* can also be captured. Some short examples follow:

- Consider a user wanting to render a MSOffice document on his smart phone. The reasoning approach could infer that this is possible through various ways, e.g. by running an appropriate Android application on his smartphone, able to read the contents of the document (e.g. Suite Office), or by running `MSWord.exe` over an appropriate emulator, or by converting the document to PDF, etc.
- As another example suppose an organization that has decided the list of acceptable or preferred file formats for archiving (this is a use case discussed in detail in Section 6.1.1). For example suppose that an

excel (`.xls`) file is not in the acceptable/preferred list of formats. However by applying a series of transformations it is possible to convert this file to an acceptable/preferred format: *"Excel (`.xls`) files can be exported to tab-delimited text files, then imported in MS Access and finally exported to comma-separated values (.csv) files"*. This is a series of two transformations: `.xls` $\xrightarrow{MSExcel}$ `.tab` $\xrightarrow{MSAccess}$ `.csv`. Having defined that we have `MS Excel` and `MS Access` the reasoning approach can infer that we can convert the `.xls` file to an acceptable/preferred file format.

– For the case of datasets, consider that we want to preserve datasets containing experimental results and would like to preserve their provenance. Suppose that for us provenance means ability to answer questions of the form: who derived the dataset, when and how this dataset was derived? We can model provenance in a way that enables checking for which datasets we have provenance and for which we have not.[1]

These capabilities enable a flexible strategy for achieving interoperability; a strategy in which standards are not mandatory. This strategy can also be more sustainable since the desired task can be achieved by combining the capabilities of existing software.

In a nutshell, the main contributions of this paper are: (a) we advance past dependency management approaches for digital preservation with converters and emulators, (b) we demonstrate how this modeling apart from capturing the preservability of converters and emulators, enables the desired reasoning regarding task performability, risk detection etc, (c) we show that with this approach we can model real converters and emulators, (d) we detail an implementation approach based on Semantic Web technologies, and (e) we present the prototype system `Epimenides` that realizes this approach. Furthermore we discuss how the proposed functionality can be applied or injected to existing systems.

The rest of this paper is organized as follows: Section 2 discusses the motivation, the context, past related works, and the key requirements regarding reasoning. Section 3 introduces the rule-based modeling approach, and provides examples showing how it can be used for modeling real converters and emulators. Section 4 discusses how the corresponding inference services can be realized using the proposed modeling. Section 5 describes the proof-of-concept system `Epimenides`, details its implementation which is founded on Semantic Web technologies, and reports various results. Section 6 discusses methods to apply the dependency management approach, methodolical issues, as well as a case study. Finally Section 7 summarizes, discusses related issues and identifies issues for further research. Some extra screenshots of `Epimenides` are given in the Appendix.[2]

[1] For reasons of space, we do not include examples for this case. We provide examples only for the case of software, since this case is in general more challenging.

2 Related Work, Background and Requirements

2.1 Related Work: Dependency Management for Digital Preservation

Preservation has been termed "interoperability with the future" where interoperability is the ability of diverse systems and organizations to work together (inter-operate). The crux of the interoperability problem is that digital objects and services have various dependencies (syntactic, semantic, etc) and we cannot achieve interoperability when the involved parties are not aware of the dependencies of the exchanged artifacts. A bit deeper, each interoperability objective can be conceived as a kind of demand for the performability of a particular *task* (or tasks). As tasks we consider actions that can be applied on a digital object (e.g. render, edit, run, compile, etc), each having its own dependencies. It is therefore evident that digital preservation is intrinsically a dependency management problem.

Regarding past works, [28] suggested to tackle digital preservation as a concrete dependency management problem and indicated how a semantic registry (compatible with OAIS[3]) can be used for offering a plethora of curation services. In the sequel [29] extended that model with *disjunctive dependencies*. Fundamental 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. A *profile* is the set of modules that are available or assumed to be available 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

[2] This paper elaborates on the ideas first presented in [32]. In comparison to that paper the current paper: presents a more expressive modeling approach (accounting for parameters and exceptions), shows how real emulators and converters can be modeled, provides implementation details, reports our experiences for implementing this approach using Semantic Web tools, presents the system `Epimenides` and reports results from its evaluation so far in the context of the ongoing APARSEN NoE.

[3] Open Archival Information System (ISO 14721:2003).

since there is not any objective method to specify exactly which are the dependencies of a particular digital object, [21] extended the model with *task-based* dependencies where the notion of *task* is used as gnomon 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 (i.e. RDF/S). Based on that model, a number of services were 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`[4] and `PreScan`[5] [20] are two systems that have been developed based on that model in the context of the EU project CASPAR[6]. Subsequently, [31] introduced a *rule-based* model which also supports task-based dependencies, and (a) simplifies the disjunctive dependencies of [29], and (b) is more expressive and flexible than [21] 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, none of the aforementioned works were able to model and exploit the capabilities offered by *converters* and *emulators*.

2.2 Other Related Works

Here we describe in brief a few related and complementary works.

File Format Identification. There are various tools for the identification of file formats. For instance DROID[7] (Digital Record Object Identification) is a tool utilising PRONOM (a system that provides information about the file formats; more in Section 6.2) for the automated batch identification of file formats. DROID identifies the format of a file based on a list of XML signatures files. Jhove[8] is another related tool that provides functions to perform format-specific identification, validation, and characterization of digital objects. Finally a tool that detects and extracts metadata from various documents is Apache Tika[9]. It offers a method that takes as input a file and returns its best guess for the type of the file, by exploiting various detection methods.

Modeling the Operational Environment of Digital Objects. TOTEM [1] is a metadata model tool that provides information about the required technical environment (software, libraries, Operation Systems and hardware components) in which a digital object can be accessed. These information could be useful for the needs of an emulation (for example finding the compatible elements e.g. hardware types compatible with a specified operation system). It has been developed in the context of KEEP[10] project. Currently it provides information about 3 technical environments. Another related model is the Preservation Network Model (PNM) [7]. PNM is a model developed in the context of CASPAR project and represents digital objects and the relationships between them. PNM describes the information which need to be packaged within an OAIS compliant Archival Information Package. [8] shows how PNMs could be used to plan preservation actions in scientific archives.

Preservation Planning. Finally we should mention Plato [2], a system for modeling various possible conversions.

The aforementioned works can assist various tasks (file format identification, getting the details about a technical environment, getting assistance in preservation planning), however none of them offers an automated reasoning for checking whether a task can be performed over a digital object, which is the ultimate objective in a digital preservation strategy.

2.3 Background: Migration and Emulation

Migration is the process of converting a digital object that runs on one platform so that it will run on another (non-obsolete) platform [14]. Its purpose is to preserve the integrity of digital objects and to retain the ability for clients to retrieve, display, and use them in the face of a constantly changing technology [36].

Emulation is generally described as imitating a certain computer platform or program on another platform or program (for a discussion see [13, 14]). It requires the creation of *emulators*, where an emulator is hardware or software or both that duplicates (or emulates) the functions of a first computer system (the guest) in a different second computer system (the host), so that the emulated behavior closely resembles the behavior of the real system. Popular examples of emulators include QEMU [4], Dioscuri [33], etc. There is currently a rising

[4] http://athena.ics.forth.gr:9090/Applications/GapManager/
[5] http://www.ics.forth.gr/isl/PreScan
[6] http://www.casparpreserves.eu/
[7] http://digital-preservation.github.io/droid/
[8] http://jhove.sourceforge.net/
[9] http://tika.apache.org/index.html
[10] http://www.keep-project.eu/ezpub2/index.php?/eng

interest on emulators for the needs of digital preservation [18]. Just indicatively, [35] 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 [33] and [23], while [33] compares applications running on Dioscuri with the same applications executed directly on the host machine). We could also mention here the KEEP (Keeping Emulation Environments Portable) project that aims at developing emulation services to enable accurate rendering of both static and dynamic digital objects. The overall aim of the project is to facilitate universal access to cultural heritage resources[11].

Another related concept is that of the Universal Virtual Computer (UVC) that was introduced in [19] (a more recent work can be found at [34]). It is a special form of emulation where a hardware and software independent platform has been implemented, where files are migrated to UVC internal representation format and where the whole platform can be easily emulated on newer computer systems. It is like an intermediate language for supporting emulation.

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 "native" 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 (as described in the running example for the case of the `game.exe`), but we should note that the emulator itself (`W4A` in running example) has its own dependencies that have to be preserved to ensure its performability (this will be made evident in Section 3.6). The same also holds for converters.

2.4 Requirements on Reasoning Services

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 fulfil 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.

Consequences of a Hypothetical Loss. 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 a user, say James, wants to compile the file `HelloWorld.cc`, his system cannot perform this task since there is not any C++Compiler. James should be informed that he should install a compiler for C++ to perform this task.

Support of Task Hierarchies. For example, if we can *edit* a text file then certainly we can *read* it. It is therefore desirable to be able to define task-type hierarchies for gaining flexibility, supporting various levels of granularity, 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.

Here 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 [11]). 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[12] or CRM Dig [27]. We focus only on 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 deriva-

[11] KEEP has created an Emulation Framework [5] (EF) which provides additional services which will help to build a more solid ground for the emulation preservation strategy. KEEP is depending on existing and future emulators, and has not created an emulator itself.

[12] http://openprovenance.org/

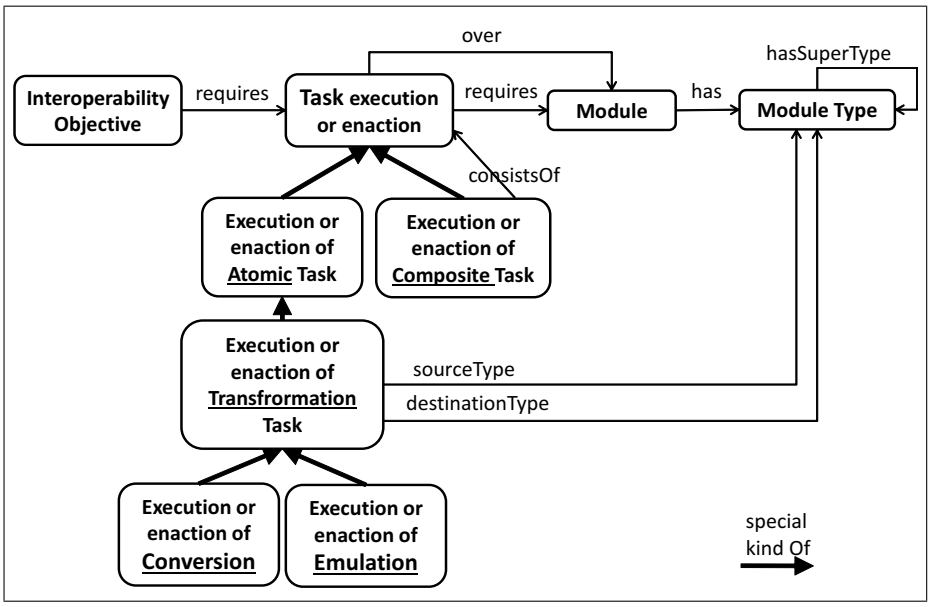

Fig. 2 Informal concept map

tion of an executable (and its compilation time or who achieved the compilation), we just care about the compiler one needs to have for future use. However, if a detailed model of the process is available, then the dependency model can be considered as a more simple and focused view of that model.

3 Modeling Tasks and their Dependencies

3.1 Overview

To assist understanding, Figure 2 depicts the basic notions in the form of a rather informal concept map. In brief, for achieving an interoperability objective we have to perform (execute or enact) one or more *tasks* over an *object* (module). In turn, to achieve performing a task over a module, we need one or more other modules. Each module has a module type, and module types can be hierarchically organized. Now conversion and emulation are special kinds of tasks each having "source" and "destination" module types (broadly speaking).

Below we will show how we can model the above, using facts (e.g. database tuples) and (Datalog) rules.

3.2 Background: Datalog

We will base our modeling and reasoning approach in *Datalog* [6] which is a query and rule language for deductive databases (syntactically subset of Prolog). We use Datalog, since it is a well known and widespread language in the community. Note that OWL [22] cannot express such rules, while SWRL [16] allows only rules with binary predicates which are not sufficient for our purposes.

In brief, a Datalog program consists of *facts*, e.g. `JavaFile(myfile.java)`, and *rules*. An example of a rule having a *head* with a predicate of two variables and a *body* with two monadic predicates is: `Compilable(X,Y) :- JavaFile(X), JavaCompiler(Y)`, which is read as follows: if we have a javafile X, and a java compiler Y, then we can infer that X is compilable by Y. 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.

3.3 Modeling Digital Objects, Type Hierarchies, and Profiles

Digital objects, e.g. digital files and their types are represented as facts using predicates that denote their types. Software components are described analogously. E.g. for our running example, described in Section 1 (Figure 1b), we can have the facts shown in the following table.

Facts	
for digital files	for software components
`PascalFile(game.pas)`	`ConverterPascal2C++(p2c++)`
	`AndroidOS(smartPhone)`
	`C++Compiler(gcc)`
	`EmulatorWinAndroid(W4A)`

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(game.pas, 2013-10-10)`.

The types of the digital files can be organized *hierarchically*, and such taxonomies can be represented with rules, e.g. to define that every `PascalFile` is also a `UTF8File` we must add the rule: `UTF8File(X) :- PascalFile(X)`.

A *profile* is a set of facts, describing the modules that are available (or assumed to be known) to a user (or a community of users).

3.4 Modeling Task-Dependencies and Task Hierarchies

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 example consider a user, say James, that holds a Java file, say `HelloWorld.java`. The predicate `Compile (HelloWorld.java)` will denote the compilability of `HelloWor ld.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/type 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)    :- JavaClassFile(X), JVM(Y).
Editable(X,Y)   :- JavaFile(X), TextEditor(Y).
```

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

```
Run(X) :- Runnable(X,Y,Z).
Runnable(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)`. Editability here presupposes knowledge of the right symbols set, the one for the intended *information object* as defined in [9].

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

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

As another example, `IntelligibleBy(X,Y) :- 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 [28] (i.e. the transitivity of dependencies).

3.5 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 step 1 of our running example, we have the file `game.pas` (which contains source code in Pascal PL), and the converter `p2c++` from Pascal to C++. Recall that Yannis has a compiler for C++. It follows that Yannis can compile `game.pas` since he can first convert it in C++ (step 1) (using the converter) and then compile it (step 2). 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).
```

The meaning of the above rule is the following : if we have a Pascal file X and a converter from Pascal to C++ Y, then we can *view* X as if it were a C++ file.

Since the profile of Yannis contains the facts `Pascal File(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).
```

3.6 Modeling Emulators

Emulation is again a special kind of task and is modeled differently. Essentially we want to express the following:

If we have :
(i) a module X which is runnable over Y, and
(ii) an emulator E of Y over Z (hosting system=Z, target system=Y

then X is runnable over Z. For example, consider the case of our running example where:

- `X=game.exe` (a file which is executable in Windows operating system),
- `Y=mycomputer` (running Windows operating system),
- `Z=smartPhone` (running Android operating system), and
- `E=W4A` (i.e. an emulator of WinOS over AndoidOS).

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 Windows OS (WinOS). For this case we would have written:

```
Run(X)   :- Runnable(X,Y).
Runnable(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(myPc)`, and by putting them together it follows that `Run(a.exe)` holds.

Now consider the running example in step 3, where Yannis has the file `game.exe` but runs AndroidOS and not a WinOS. However Yannis has the emulator `W4A` (i.e. an emulator of WinOS over AndoidOS). The profile of Yannis contains:

```
WinExecutable(game.exe)
AndroidOS(smartPhone)
EmulatorWinAndroid(W4A)
```

To achieve our goal (i.e. to infer that `game.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 `runnable` 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 `Runnable`
This relies on adding the following rule:

```
Runnable(X,Y,Z):- WinExecutable(X),
                  EmulatorWinAndroid(Y), AndroidOS(Z).
```

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

Note that here we added a rule for the `runnable` which has 3 variables signifying the ternary relationship between executable, 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 operate 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 profile of the user has an emulator of type `EmulatorWinAndroid` (here `W4A`) and `smart Phone` is of type `AndroidOS`, then that rule will infer that `WinOS(smartPhone)`, implying that the file `game. 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 introducing 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 emulators are modules that can have their own dependencies, they should be runnable in the hosting system. To ensure their runnability during an emulation we have to update the above rules as follows (notice the 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)         |
```

3.6.1 Modeling Important Parameters

Sometimes it is important to model the required (important) parameters for the performability of a task. For example, an emulator may need a particular parameter for emulating a particular system. In this case it is beneficial to model this explicitly. Methodologically, it is not suggested to model all parameters, e.g. those of minor importance, but only the crucial ones, those for enabling the required reasoning. For example consider the following rule:

```
WinOS(X):- AndroidOS(X), EmulatorWinAndroid(Y),
           Runnable(Y,X)
```

and suppose that this emulator needs one particular parameter for emulating windows, say a file `winImg.dat`. One way to capture this, is to extend the above rule as:

```
WinOS(X):- AndroidOS(X), EmulatorWinAndroid(Y),
           Runnable(Y,X), Module(winImg.dat)
```

where `Module` is the top class of the module type hierarchy. This rule will fire only if the `winImg.dat` exists in the user's profile.

3.6.2 Handling Exceptions or Special Cases

Suppose that we know that a given Windows application, can run on Android using a Windows emulator, but this is not true for every Windows application. For example we might know that the emulator `W4A` cannot run the application `calendar.exe`. To tackle this situation we can add the fact `Exception(calendar.exe,W4A)`, which has a negative interpretation and is read as follows: *the emulator W4A cannot emulate the application calendar.exe*. Now we can extend the approach A as follows:

```
A'':Runnable(X,Y,Z):-
     WinExecutable(X),
     EmulatorWinAndroid(Y),
     AndroidOS(Z),
     NOT Exception(X,Y),
     Runnable(Y,Z)
```

The difference between the old rule is that we have added a negated `Exception(X,Y)`, meaning that the application X is runnable if the atom `Exception(X,Y)` is false. Updates are not problematic, e.g. suppose that a new patch is distributed that can make `calendar.exe` to work. In that case, one could simply delete the fact `Exception(calendar.exe,W4A)` without updating the body of the rule.

Let's now consider an exception that concerns a class of modules. In such a case one might think of defining a rule to specify these extensions, e.g. `Exception(X,Y) :- WinInternal(X), EmulatorWinAndroid(Y).` Note that this will work also for the types that refine `WinInternal` as well as for the types that refine `EmulatorWinAndroid`. However this approach makes hard the management of patches. To make this clear, suppose that a patch that solves the problem only for one particular module, say `m1`, of type `WinInternal`, comes up. Due to the previous rule we will get `Exception(m1, W4A)` and notice that this is an inferred fact that we cannot delete. This is a complex evolution scenario, that would require defining update operations similar in kind with the operations of *contraction* and *disassociation* as defined in [26]. Therefore it is not suggested to use rules for specifying exceptions. Instead they should be defined by facts.

Finally note that the approach B cannot be extended analogously as previous for the approach A, because approach B by default states (as we have seen in Section 3.6) that we can emulate the entire system.

3.7 Synopsis of the Modeling Approach

To synopsize, methodologically 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 (which is usually binary) for denoting the dependencies of task (e.g. `Read` and `Readable`, `Run` and `Runnable`). 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). 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 (approach A), or write a rule for classifying the system that is equipped with the emulator to the type of the emulated system (approach B). Also, 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). Finally, we can capture special cases (e.g. exceptions, modeling of parameters) accordingly.

3.8 Examples of Modeling Real Converters and Emulators

To evaluate the adequacy of the proposed modeling approach, in this section we show how some well known converters and emulators can be modeled using our approach.

Texi2HTML converter: Texi2HTML[13] is a Perl script, which converts Texinfo source files to HTML output. Texinfo is the official documentation format of the GNU project.
To model this scenario we must introduce classes for the various module types, i.e. for texi files, for perl scripts, for perl interpreters, and for the particular converter (from texi to HTML). For instance, consider a user who has a `myfile.texi` file, the `strawberry-perl.exe` perl

[13] http://www.nongnu.org/texi2html/

interpreter, and the `Texi2htmlScript.pl` converter (from texi to HTML). The profile of this user will contain the facts:

```
PerlScript(Texi2htmlScript.pl)
PerlInterpreter(strawberry-perl.exe)
TexinfoFile(myfile.texi)
Texi2HTMLConverter(Texi2htmlScript.pl)
```

Firstly we have to declare a rule for the conversion. In this case the rule will be:

```
HTML(X) :- TexinfoFile(X), Texi2HTMLConverter(Y),
           Run(Y)
```

Note also that `Texi2htmlScript.pl` (as any perl script) requires the availability of a Perl interpreter to run, therefore we should add the following rule to ensure its runability:

```
Runnable(X,Y) :- PerlScript(X), PerlInterpreter(Y)
```

Dioscuri emulator: Dioscuri[14] is a component-based x86 computer hardware emulator written in Java. Each hardware component is emulated by a software surrogate called a module. By combining several modules the user can configure any computer system, as long as these modules are compatible.
For example consider a user having dioscuri emulator version 0.7.0 (which requires a JVM to run) and suppose he wants to run `Chess.exe`, a 16-bit DOS Application on his computer with the Windows XP Operating System (`jre1.5win` installed).
Declaring again the appropriate classes, the profile of this user will contain the facts :

```
DOSExecutable(Chess.exe)
WindowsXPOS(mycomputer)
DioscuriEmulator(dioscuri-0.7.0.jar)
JavaByteCode(dioscuri-0.7.0.jar)
```

The execution of a Java ByteCode requires a JVM so:

```
Runnable(X,Y) :- JavaByteCode(X), JVM(Y)
```

According to Section 3 now we can write the rule for the emulation:

```
DOSOS(X) :- WindowsXPOS(X), DioscuriEmulator(Y),
            Runnable(Y,X)
```

.

QEMU emulator: QEMU[15] is a generic open source machine emulator and virtualizer that can run a target operating system. To emulate another machine one needs to have the process emulator (QEMU) and an ISO image of the machine he wants to emulate. For instance, consider a user who wants to emulate Windows XP on his machine running Linux, by providing the appropriate ISO image (e.g. `WinXP.iso`). His profile will contain the the following facts:

```
LinuxOS(mycomputer)
QEMUEmulator(QEMU1.1)
ISOFile(WinXP.iso)
```

As we have stated at Section 3, apart from declaring the rule for the emulation, we have to ensure that the emulator is runnable in the hosting system (here `mycomputer`), therefore we have to include also a `Runnable` rule. The resulted rules will be:

```
Runnable(X,Y) :- QEMUEmulator(X), LinuxOS(Y)
WindowsXPOS(X) :- LinuxOS(X), QEMUEmulator(Y)
                  Module(WinXP.iso), Runnable(Y,X)
```

Notice that the user in his profile has the fact `ISOFile(WinXP.iso)`, but the above rule uses the atom `Module(WinXP.iso)`. The rule will fire because `Module` is the top class of the module type hierarchy (i.e. if something belongs to the class `ISOFile` then it also belongs to the class `Module`).

4 Reasoning Services

In general, Datalog query answering and methods of logical inference can be exploited for enabling the required inference services (performability, consequences of a hypothetical loss, etc). Here we describe how the reasoning services described in Section 2 can be realized using the proposed modeling approach and framework.

Task-Performability Checking.
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.

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.

For example let us check the performability of the running example, described in Section 1, for the user Yannis. The goal is to check if Yannis can run the `game.pas` file on his mobile phone. Indeed the fact `Runnable(game.pas ,smartPhone)` can be derived as shown in the proof tree of Figure 3. In that figure the facts are represented by a rectangle, while the greyed rectangles show the applicable rules. The used facts in this example are:

```
PascalFile(game.pas),
ConverterPascal2C++(p2c++),
WinOS(mycomputer),
AndroidOS(smartPhone),
```

[14] http://dioscuri.sourceforge.net/
[15] http://wiki.qemu.org/Main_Page

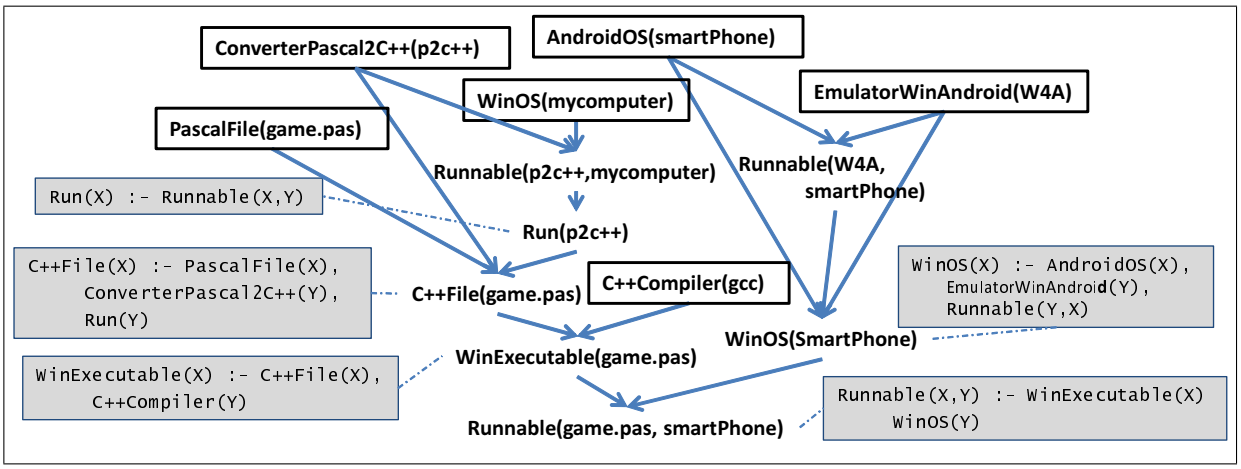

Fig. 3 The "proof tree" of the running example, showing how we can get `Runnable(game.pas, smartPhone)`.

```
C++Compiler(gcc),
EmulatorWinAndroid(W4A)
```

Consequences of a Hypothetical Loss.
Suppose that we want to identify the consequences 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 returned set of elements[16], and (c) we compute and return the elements in $A \setminus B$ (they are the ones that will be affected).

Computation of Gaps (Missing Modules).
The gap is actually the set of facts that are missing and are needed to perform a task. There can be more than one ways 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. if we want to execute a Windows application we have to do it on a Windows machine or use an appropriate emulator or an appropriate converter). One method to fill the gaps is to construct and visualize an AND-OR graph that contains information about only the related facts and rules. Such an approach is described in [32]. An alternative (or complementary) approach is to allow the user to *gradually* explore the possibilities and navigate through the possible paths. The implemented system that is described in Section 5 follows the last approach.

[16] In an implementation over Prolog, we could use the *retract* feature to delete a fact from the database.

5 Epimenides: A Proof-of-Concept System

For proving the technical feasibility, as well as for demonstration and dissemination purposes, we have build a web accessible system, called `Epimenides`[17].

5.1 Use Cases and User Interface

The Use Case diagram that provides an overview of the supported functionality is given in Figure 4. The application can be used by several users, that can build and maintain their own profiles. To be flexible, a gradual method for the definition of the profiles is supported. The Knowledge Base (KB) of this system currently contains 2,225 RDF triples and it is described in Section 5.3, while the main scenario is described next.

After logging in, *the user can upload a digital object* (a single atomic file or a compressed bundle of files) and select the task for checking its performability. The system then checks the dependencies and identifies the tasks that can be executed, as well as the missing resources for performing certain tasks.

The curator can define new tasks to the system. After uploading a file, its type is identified by exploiting its file extension (alternatively other tools that analyze the contents of files like JHove[18] or JMimeMagic[19] can be exploited). The KB contains the dependencies for some widely-used types, therefore the appropriate task-based dependencies are shown to the user. The user can then *add* those that (s)he already has, and this is actually the method for defining his profile *gradually*. In this way he does not have to define his profile in one

[17] http://www.ics.forth.gr/isl/epimenides/
[18] http://jhove.sourceforge.net
[19] http://jmimemagic.sourceforge.net

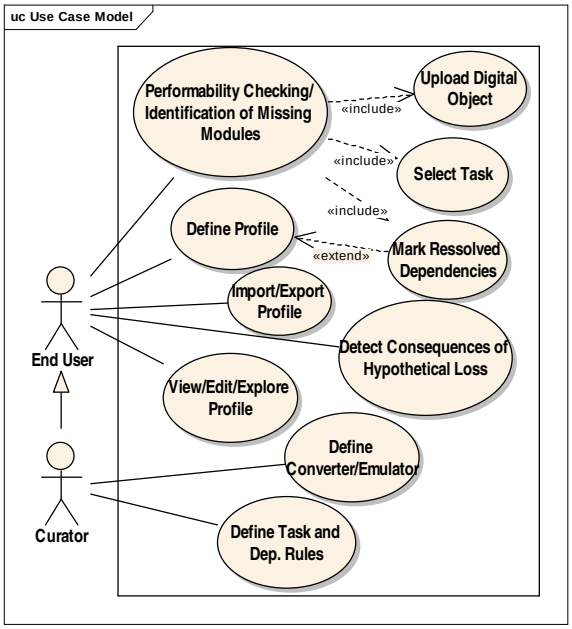

Fig. 4 Use Case Diagram of `Epimenides`

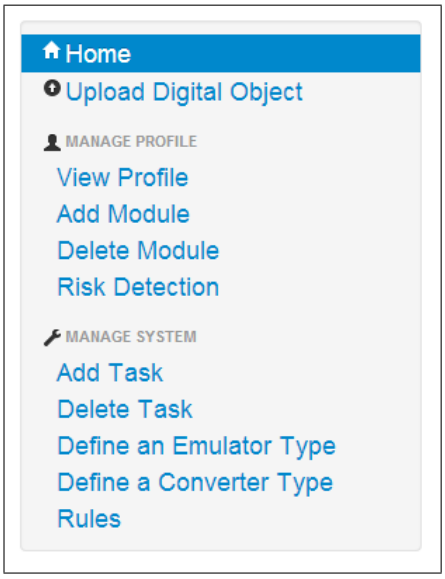

Fig. 5 Main functionalities of `Epimenides`

shot. The system stores the modules of each user (those modules marked as "I have them") to the RDF storage. The profiles are stored using different graph spaces and a user can export a profile, or import a profile. Also note that a user can maintain more than one profiles.

User Interface

The user interface contains a menu divided in three sections as shown in Figure 5. The first contains the main option of the application: "Upload Digital Object". The "MANAGE PROFILE" section contains options available to any (simple) user. The user can also add/delete modules to his profile as well as exporting his profile or importing the profile of a different user.

The "MANAGE SYSTEM" section contains options for a curator user. Such a user has also the ability to define Tasks, Emulators and Converters. To properly add a Task, an Emulator or a Converter one has to provide extra information from which the application will produce the required rules (as we will describe in Sections 5.4 and 5.5). A simple user can add to his profile an emulator X, only if it has been properly defined from a curator (and consequently the application has produced the required rules).

Figure 6 shows an indicative screenshot of `Epimenides`. This figure shows the result analysis of three uploaded digital objects. For the first file it is being reported that the selected task (i.e. Rendenring) cannot be performed over that digital object). While in contrast, for the last two files, the selected tasks can be performed successfully. More screenshots of the application are provided in Appendix A.

5.2 The RDF/S Layer (for the core functionality)

The server side of `Epimenides` uses Semantic Web technologies. It uses the *OpenLink Virtuoso* as the underlying triple store, which is a general purpose RDF triple store with extensive SPARQL and RDF support [12]. 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 because 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;

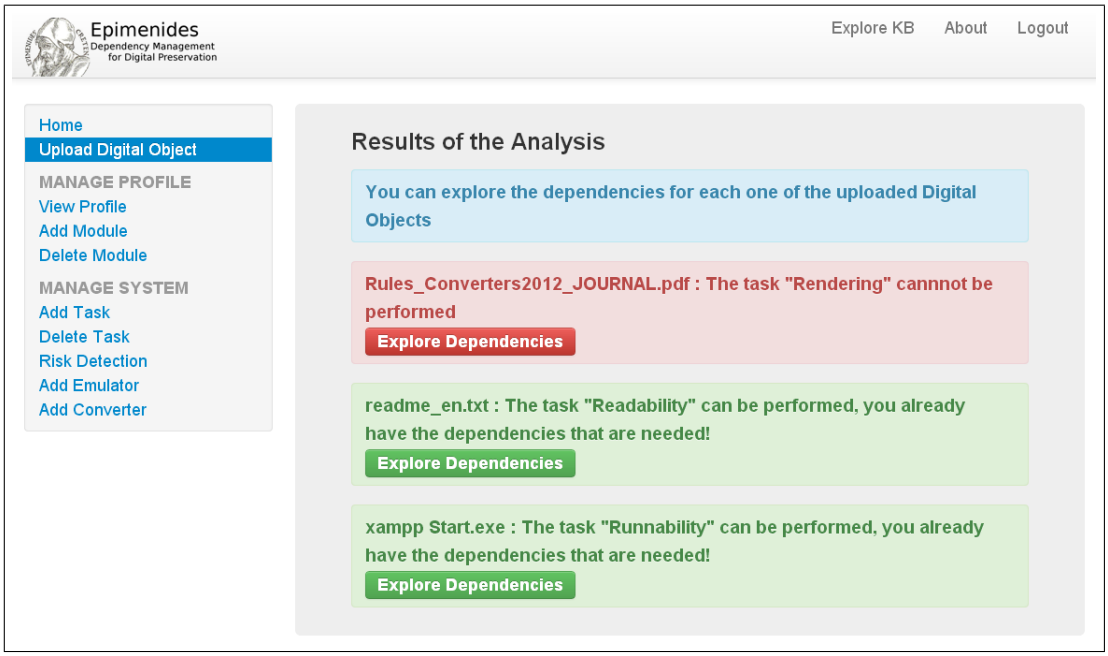

Fig. 6 Results of the analysis

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` relationship. For example the fact `PascalFile(game.pas)` and the rule for defining the module type hierarchy `TextFile(X) :- PascalFile(X)` will be expressed using the following quads:

```
g, <PascalFile>, rdf:type, rdfs:Class
g, <TextFile>, rdf:type, rdfs:Class
g, <PascalFile>, rdfs:subClassOf, <TextFile>
g, <game.pas>, rdf:type, <PascalFile>
```

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

```
<yGrph>, <game.pas>, rdf:type, <PascalFile>
<yGrph>, <p2c++>, rdf:type, <ConverterPascal2C++>
<yGrph>, <gcc>, rdf:type, <C++Compiler>
<jGrph>, <HelloWorld.java>, rdf:type, <JavaFile>
<jGrph>, <javac_1_6>, rdf:type, <JavaCompiler>
<jGrph>, <NotePad>, rdf:type, <TextEditor>
```

Dependencies: The rules regarding the performability 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>}
```

To capture the compilability of other kinds of source files (i.e. C++, pascal etc.) we extend the previous statement using the UNION keyword (this is in accordance with the Datalog-based rules; multiple rules with the same head have union semantics). For example for the case of Java and C++ is captured by:

```
CONSTRUCT{?x <compilable> ?y}
WHERE{
       {?x rdf:type <JavaFile>.
        ?y rdf:type <JavaCompiler>}
       UNION
       {?x rdf:type <C++File>.
        ?y rdf:type <C++Compiler>}
}
```

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>}
```

Note the last condition refers in an inferred type triple (Runnable). If there are more than one converters that change modules to a specific module type then the construct statement is extended using several WHERE clauses separated by UNIONs, as shown previously.

Emulators: Consider the 3^{rd} step of the running example, described in Section 1, i.e. a user wanting to run `game.exe` upon his Android operating system. The approach B can be expressed by:

```
CONSTRUCT{?x rdf:type <WindowsOS>}
WHERE{?x rdf:type <AndroidOS>.
      ?y rdf:type <EmulatorWin4Android>.
      ?y <runnable> ?x}
```

If the emulator needs a particular parameter, as for example the module `winImg.dat` as described in Section 3.6.1 we have to add an extra triple on the previous query for this module, so we model the emulator as :

```
CONSTRUCT{?x rdf:type <WindowsOS>}
WHERE{?x rdf:type <AndroidOS>.
      ?y rdf:type <EmulatorWin4Android>.
      ?y <runnable> ?x.
      <winImg.dat> rdf:type <Module>}
```

To support Exceptions (described in Section 3.6.2) we have to define an INSERT query. For example to declare that the emulator `W4A` cannot emulate the `calender .exe` we have to perform the query:

```
INSERT{<calendar.exe> <exception> <W4A>}
```

Consequently we should change the CONSTRUCT query to take into account the possible exception by exploiting the NOT EXISTS functionality of SPARQL as follows:

```
CONSTRUCT{?x <runnable> ?y}
WHERE{
    ?x rdf:type <WinInternal>.
    ?z rdf:type <EmulatorWin4Android>.
    ?y rdf:type <AndroidOS>
    FILTER NOT EXISTS{?x <exception> ?z}
}
```

Services: To realize the reasoning services (e.g. task performability, consequences of a hypothetical loss, etc), we rely on SPARQL queries. For example, to answer if the file `HelloWorld.java` can be compiled, we can 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 "Consequences of a Hypothetical Loss" service requires SELECT and DELETE SPARQL queries (as discussed in 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 $A \setminus B$ will reveal the modules that will be affected. If empty this means that there will be no risk in deleting `Notepad`.

5.3 The Knowledge Base of `Epimenides`

In Section 5.2 we described the core part of the modeling and reasoning approach system. For an operational system we have to tackle various other aspects which we detail here.

Figure 7 shows the architecture of the system's KB. As we have mentioned before, the profile of a user contains triples with the modules that hold on his system, while the application contains information about the dependencies that are needed to execute a task. For the representation of the modules the KB contains all the MIME media types[20] expressed as a `subClassOf` hierarchy (part of this hierarchy is shown in the left of Figure 8). The dependency rules are also stored in the KB as strings of SPARQL queries. Finally the KB

[20] Multipurpose Internet Mail Extensions (MIME) is an Internet standard that extends the format of email.

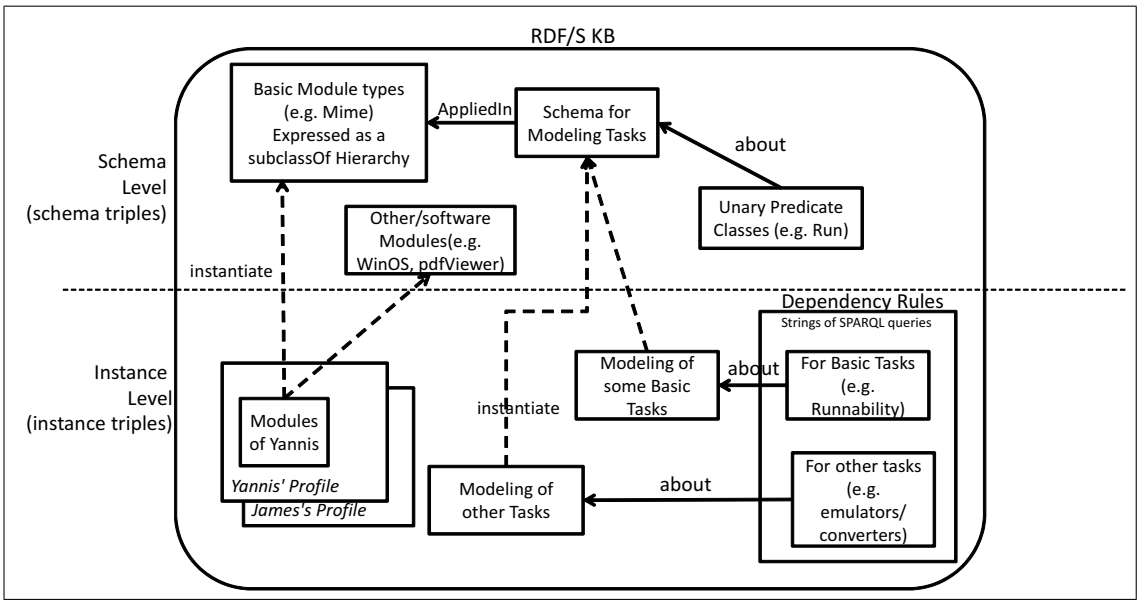

Fig. 7 Architecture of the KB

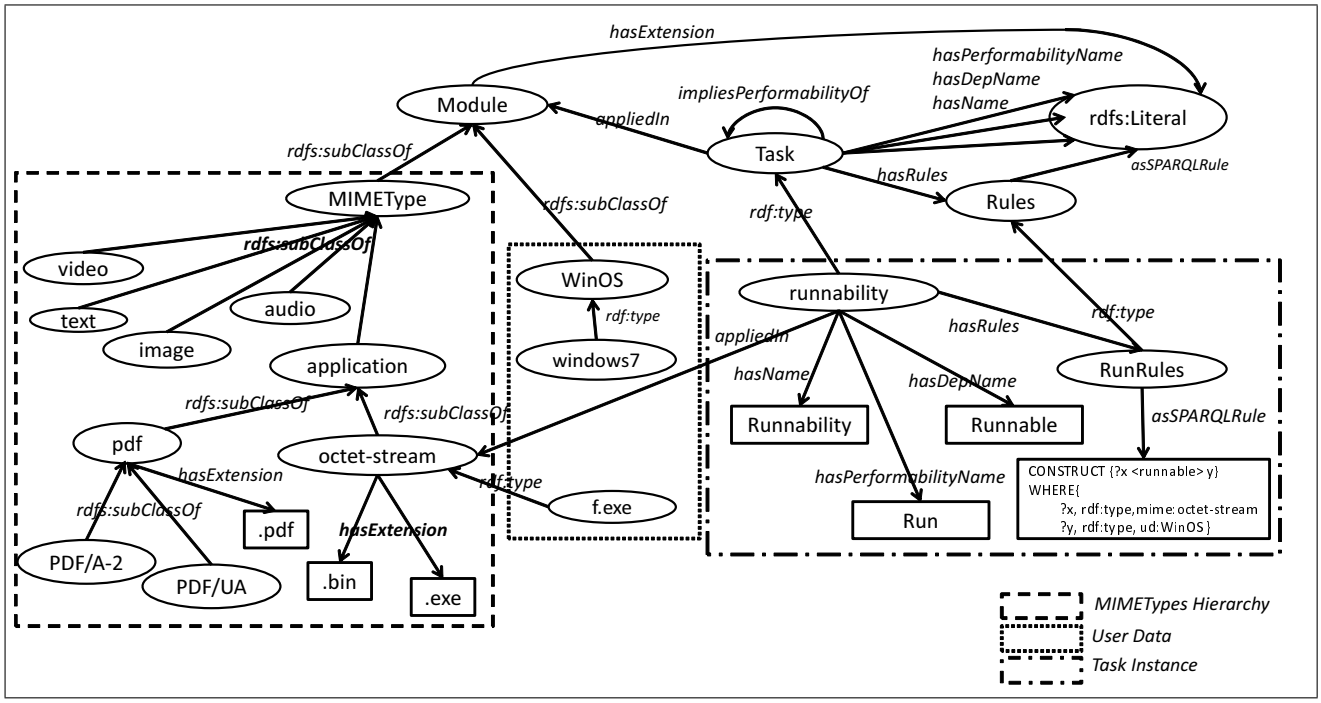

Fig. 8 The contents of an RDF/S KB that follows the architecture of Figure 7

also contains information about tasks. Note that any resource in the KB is represented by a unique URI (Uniform Resource Identifier). Different versions of MIME Type files can be supported (each of these versions has a different URI) by extended the MIME-type hierarchy. Figure 8 shows an example where different versions of pdf files (PDF/A-2, PDF/UA) can be supported.

To explain the structure of the KB we shall use an example that is illustrated in Figure 8. Suppose a user that his profile contains only the module *WinOS* and he uses the application for first time. The user uploads a file, say $f.exe$, and the system by its filetype extension or by analyzing the contents using tools like those that have described in Section 2.2 , can realize that the uploaded file is an executable file, and that

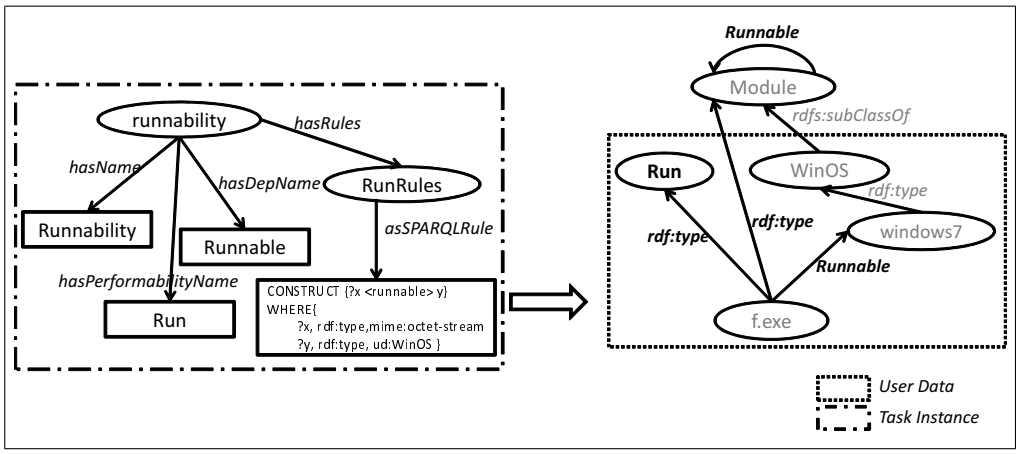

Fig. 9 Operational KB

it has the `"application/octet-stream"` MIME type and consequently it belongs to the `octet-stream` class of the KB (as shown in Figure 8). To achieve this for the first case (using the file extension) any MIME type class in the KB has the property `hasExtension`. In this way the KB contains the triple `(<octet-stream>, <hasExtension>, ".exe")`. This means that the system from the extension understands the class that models the file type by running the SPARQL query:

```
SELECT ?className
WHERE{?className <hasExtension> ".exe"}
```

In the second case (e.g. using *Jhove*) the system recognizes directly this class.

By knowing the class that models the type of f, the system can find the tasks that can be applied to the uploaded file by the property `appliedIn`, which has domain a task and range a MIME type. The system shows a list of all these tasks, returned by the following query:

```
SELECT ?taskName
WHERE{?className <hasExtension> ".exe".
      ?task <appliedIn> ?className.
      ?task <hasName> ?taskName}
```

The user can select one of the retrieved tasks and the next step for the system is to check if this task can be performed. This can be done by the task-performability service (as we have described in Section 4). To implement this service we have added in the class `Task`, the following predicates (also shown in Figure 8):

- `hasName` (literal) - the name of the task e.g. *Runnability*
- `hasPerformabilityName` (literal) - the unary predicate that denotes the performability of the task e.g. *Run*
- `hasDepName` (literal) - the binary predicate that denotes the dependencies of task e.g. *runnable*
- `hasRules` (Rules) - the rules that determine when the task can be performed.

To enable the performability checking service, the system produces the *Operational KB* in which new classes and new properties are created and populated. The name of the new classes and properties are determined by the properties `hasPerformabilityName` and `hasDepName` respectively. The *Operational KB*, for short *OKB*, is a superset of KB. Specifically, it contains the results of the application of *all* rules that the KB contains. Whenever the KB changes (e.g. user uploads a new file or a new rule is applied) the OKB is updated, and all rules are applied again. In this way, query answering can indeed support the desired services for task performability, taking also into account the emulators and the converters.

In our case suppose that the user has selected the task with name "*Runnability*". All the rules of the KB are applied and the OKB is created. Figure 9 depicts the OKB for our example; its right side shows how the *User Data* are changed when the OKB is produced (the newly produced data are shown in bold).

The class `Run` and the property `Runnable` have already been created in the OKB. Now the system using the property `hasPerformabilityName`, issues the query `Run(f)` in the profile of the user (in OKB). Obviously, the answer of the query in our example is true, as shown in Figure 9, therefore the system informs the user that this task can be performed. In the case where the selected task could not be performed, the system would inform the user for the missing dependencies.

To determine the dependencies that are *missing*, and are required for performing the selected task (the Computation of Gaps service as described in section 4), the system uses the Dependency Rules that are stored

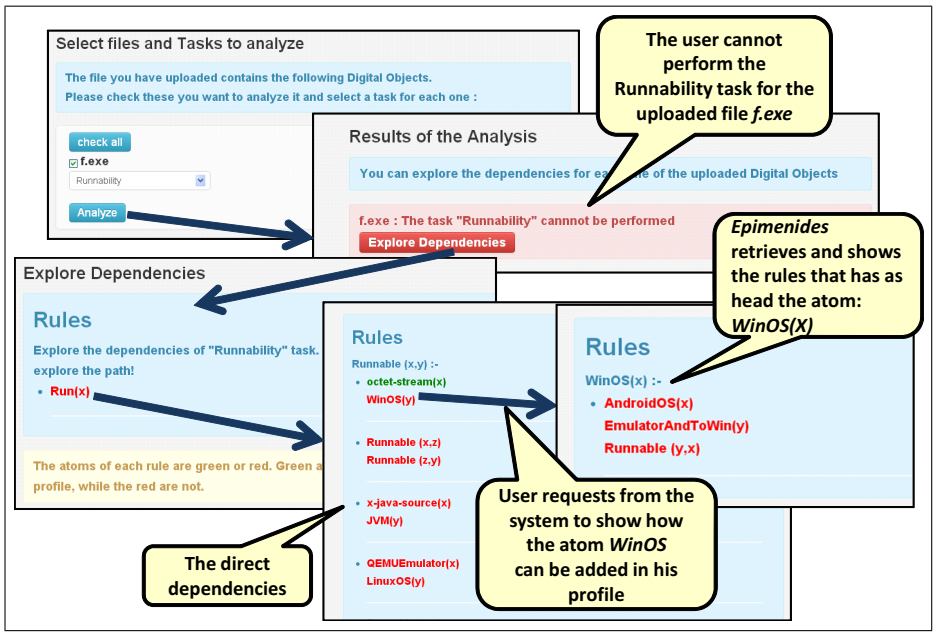

Fig. 10 The Gradual Expansion in `Epimenides`

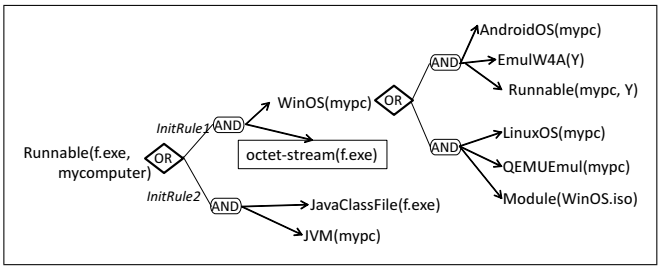

Fig. 11 Gradual Expansion

in the KB. Specifically, for each rule of the selected task, it retrieves, through the property `asSPARQLRule`, the direct dependencies, which are actually a set of atoms. These atoms are shown to the user and he can ask the system to show how an atom can be satisfied. In this case the system explores the KB for rules (including rules for emulators/converters) that have as head the selected atom. The above procedure can be repeated for the new rules, and so on. In this way a gradual expansion is created as the user gradually explores the possible paths.

Figure 11 shows an example of the above procedure. This example corresponds to the case where the user cannot perform the `Runnability` task for the uploaded file *f.exe*. The system retrieves and shows to the user the direct dependencies as specified by two rules denoted by `InitRule1` and `IntitRule2`. This means that to turn *f.exe* runnable either `InitRule1` OR `InitRule2` should be satisfied. The body atoms of each rule are shown, e.g. `InitRule1` requires a `WinOS` AND an `octet-stream` file. Subsequently the user can request from the system to show how each of these atoms, say `WinOS(mycomputer)` of `InitRule1`, can be satisfied and/or added to his profile. Analogously, the system retrieves and shows the rules that have as head the atom `WinOS(X)`.

This process is supported also by `Epimenides`, and Figure 10 shows a series of screenshots that illustrate it. We should also mention that during the above process the system informs the user about those atoms which are already satisfied by his profile. This assists the user to decide which paths are useful to to explore. In Figure 11 these atoms are enclosed in rectangle, while `Epimenides` colors them green (see Figure 10).

5.3.1 The Current KB

Currently the KB (Knowledge Base) of `Epimenides` contains 657 Module types, including 647 MIME type Modules. For each of the 647 Mime types the KB contains extra information (e.g. the extension of a MIME type). The KB also contains information about three tasks (readability, runnability, rendering), and for their dependencies it contains 24 rules. In total, the KB contains 2,225 RDF triples. Of course, as we have already mentioned, a user can enrich the knowledge base by adding his/her own Module Types, Tasks and Rules using the GUI of `Epimenides`.

Note that as we can see in Figure 12 the KB is composed of two different graph spaces: the user profile graph space that contains information for the modules that a user holds, and the system's graph space that contains information about the rules, the tasks and the modules.

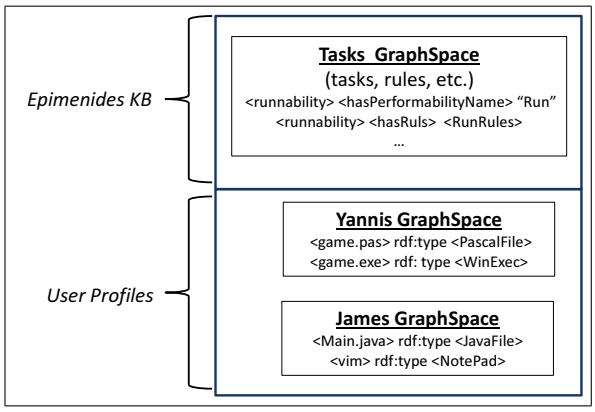

Fig. 12 Graph spaces of KB

5.4 Aiding the Ingestion of Tasks

Above we have described the use cases for end users. To assist the job of a curator, we have provided a simple method for adding new tasks and for modeling their dependencies (this is related to the use case named "Define Task and Dep. Rules" in Figure 4). Specifically he only has to provide some input and the application produces the required rules. Specifically the curator should provide:

- The unary predicate that denotes the performability of the task (e.g. Edit),
- the MIME type/s of the digital objects (e.g. text/plain) over which this task can be applied,
- the module/s that is/are required for the selected MIME Type (e.g. Text Editor).

In this way, anyone can define new tasks, without having to be aware of the internals of this approach.

5.5 Aiding the Ingestion of Converters and Emulators

Analogously, we have made simple the definition of new emulators. This is related to the use case named "Define Converter/Emulator" in Figure 4. In brief, to define an emulator the user/curator has to provide:

- the name and the URI (or just filepath) of the emulator that we want to register, let denote this by E and $Epath$ respectively,
- what this emulator emulates (e.g. the WindowsXp Operating System), let denote this input by A,
- in what System the emulator runs (e.g. Linux Operating System), let denote this input by B,
- Files or other modules that the emulator uses (e.g. ISOFile), and let denote such input by $P_1, \ldots, P_k$.

Using the above input, the system automatically produces what is required (in terms of facts and rules) based on the method described in Section 3. Specifically,

- let A_{pred} denote the predicate corresponding to A, e.g. the URI of the RDF class WindowsXP
- let B_{pred} denote the predicate corresponding to B, e.g. the URI of the RDF class LinuxOS
- regarding $P_1, \ldots, P_k$, and assuming that each one of these is specified by a URI, they will be used for extending the emulator rules (as described earlier).

At first the system creates a new RDF class E and the fact $E(Epath)$. Subsequently, it creates the rule:

```
A_pred(X):- B_pred(X), E_pred(Y), Runnable(Y,X),
            Module(P_1), ..., Module(P_k).
```

As regards the runnability of the emulator, the system creates the rule:

```
Runnable(X,Y) :- E_pred(X), B_pred(Y)
```

5.6 Query and Reasoning Efficiency

Currently `Epimenides` uses the *OpenLink Virtuoso* RDF triple store. Below we report a few indicative times that concern the current knowledge base of `Epimenides`[21] which contains 2,225 RDF triples and 23 rules. We should clarify that the times are indicative since we did not focus neither on efficiency, nor on any kind of optimization. In general performance depends on the capabilities of the adopted triplestore used (for a comparative analysis see [15]).

Creation Time for the KB. The time to create from scratch the entire KB by loading N-Triple files requires about 4 seconds. The time to add or delete a module is negligible.

Creation Time for the OKB. As we have already mentioned in Section 5.3, the *OKB* is produced by "firing" the rules which in turn produce new triples. The time for the creation of the OKB depends on the number of the fired rules. Table 1 shows the time required for creation the OKB from KB, based on the number of the fired rules. We can see (forth column) that this process takes at most 12 seconds (what each row means is explained below).

Task Performability Checking. The service for task performability checking relies on the *OKB* and it is reduced to plain query answering. Therefore it is very fast, in average it takes around 37 milliseconds.

Consequences of a Hypothetical Loss. We have

[21] The experiments were carried out using the Virtuoso 06.01.3127 version, running in a DualCore linux machine with 3GB RAM.

# of deleted Modules	# of Fired Rules	Size (in triples) of the Produced *OKB*	Time for Producing the *OKB*	Overall Time for the "Consequences of a Hypothetical Loss" service
3	2	2,227	3.32 sec	6.14 sec
5	6	2,245	7.67 sec	13.21 sec
12	14	2,261	8.90 sec	14.49 sec
18	24	2,299	11.40 sec	19.81 sec

Table 1 Some indicative measurements of time

also made some experiments regarding the time required for the "Consequences of a Hypothetical Loss" service. This is a composite task that includes all the previous steps, therefore it is appropriate for experimentation or for benchmarking one particular implementation (or underlying triplestore).

Specifically we measured the time required for identifying the consequences on the peformability of a task t_i after removing one module m_j or more. In this experiment we used 3 tasks $\{t_1, t_2, t_3\}$, and studied the case of removing various numbers of modules. Specifically in each case we were removing the modules $\{m_1, \ldots, m_y\}$, for the following values of y: $3, 5, 12, 18$. Recall that the consequences of a hypothetical loss is achieved by (a) computing the answer of the query t_i(X) (let A be the returned set of elements), (b) deleting m_j from the database, reconstructing the OKB and answering again the query t_i(X) (let B be the returned set of elements), and (c) computing and returning the elements in $A \setminus B$ (they are the ones that will be affected).

The above procedure requires the construction of the OKB twice (one before the computation of A and one before the computation of B). Table 1 reports execution times. We can see that the times range from 6 to 20 seconds.

Conclusions. In general we do not expect any difficulty in achieving efficiency (mainly because task performability is reduced to plain query answering over the OKB which has already materialized the required information). Furthermore, if the adopted triplestore supports custom backwards reasoning, then one will not even have to produce the OKB, which is the most expensive task.

5.7 Evaluating Usability

We decided to evaluate the usability of the system for investigating if a user can understand the main concepts of the approach by using the system, and how the system per se is usable. For this reason we created a short tutorial for the system[22], we defined some scenarios[23] that we asked users to carry out using the system, and we prepared a small questionnaire that the users had to answer after using the system.

Ten users answered this questionnaire with ages ranging from 20 to 30. All of the participants had a computer science background (some of them had a MSc in Computer Science). We can partition the participants in two groups: the advanced group consisting of 3 users (from APARSEN NoE[24]) and the regular ones consisting of 7 users (from Computer Science Department, University of Crete). The advanced users were aware of the dependency management approach, while the regular ones were not. For this reason, and before starting the evaluation, we gave to each regular user a brief tutorial on using the system through examples.

Below we summarize the answers of the questionnaire (detailed results are given in Figure 13). The results showed that 90% of the participants completed the scenario A, while scenario B was completed from all users (100%). The time to complete both scenarios A and B was less than 6 minutes. From the above we can conclude that `Epimenides` is understandable and easy to use. In questions 5 and 6, all users (100%) answered that the system assisted them in checking the performability of a task and that they better understood why a task can be performed in an existing and unknown profile. This demonstrates the value of the system. Finally, 70% declared that this application is useful for an organization with a big dataset of digital objects. It is also worth noting that no user had ever used any relevant system. At last, a big percentage (90%) of the participants rated with 3 (high) the potential of this approach.

6 Applicability

There are more than one ways to apply the approach presented in this paper. Below we discuss two main approaches.

[22] `http://users.ics.forth.gr/~kargakis/data/demoUsersGuide.pdf`

[23] `http://users.ics.forth.gr/~kargakis/data/UserExperience.pdf`

[24] http://www.alliancepermanentaccess.org/

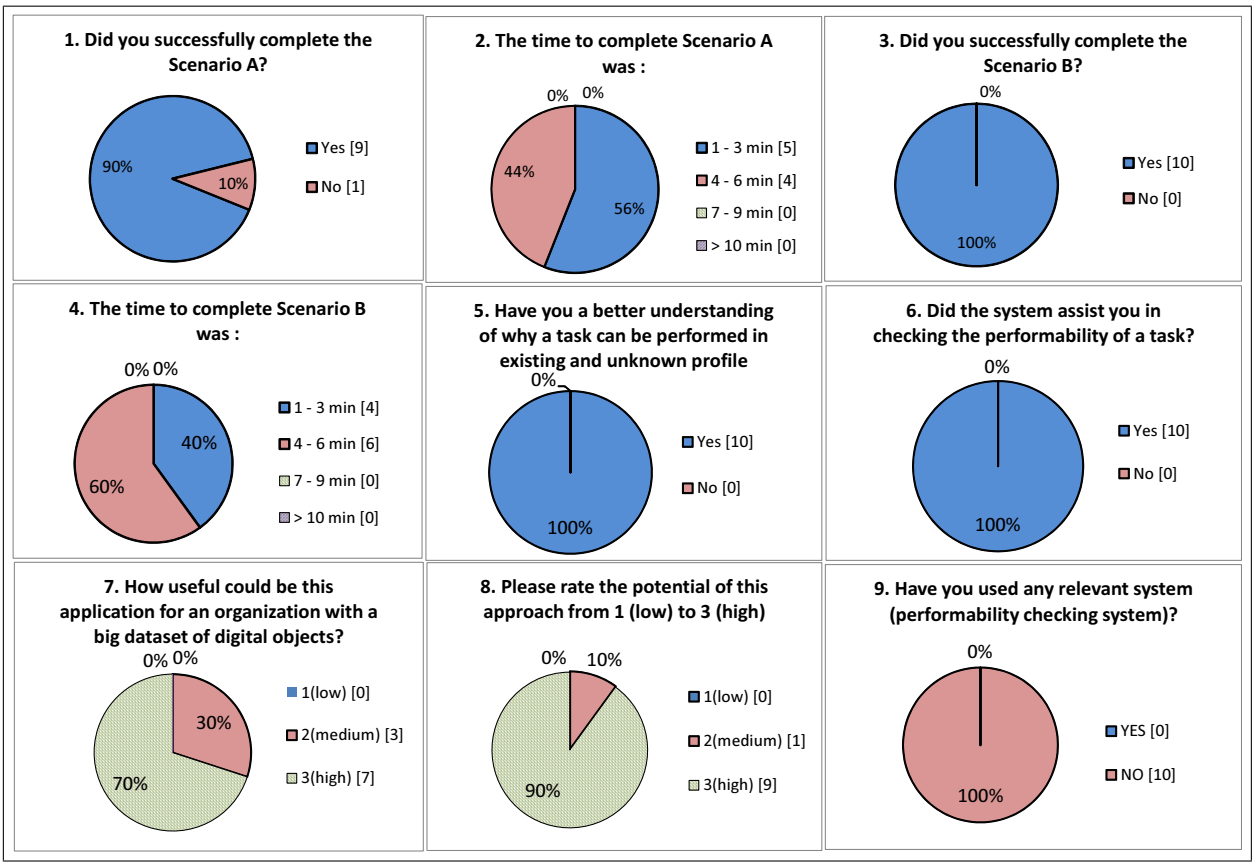

Fig. 13 Analysis of the responses to the questionnaire

– **As a System.** Here the idea is to have one dedicated system, adapted to the needs, practices and other operational system of the organization/archive. Systems like `Epimenides` fall into this case. Alternatively, one could "inject" the dependency management approach to existing repository management systems. For instance, Fedora[25] is a widely used repository management system for digital objects that provides tools and interfaces for the creation, ingest, management, and dissemination of content. The dependency management approach could be implemented by extending the Fedora repository. This could be quite straightforward since the Fedora stores metadata using RDF/S and the set of relations that can be used for connecting objects is not limited.
– **As a Service.** The proposed approaches could be offered as a service by third party providers, e.g. by a provider of cloud services who apart from offering storage services, it offers various virtualization services and uses the methodology and techniques described in this document for realizing them. These services could be based on the Service Oriented Architecture (SOA) since virtualization and service composition are some of the key features of SOA. The adoption of SOA could bring benefits in developing new services by combining others, and in achieving scalability. We could say that the long term vision is the virtualization of the basic preservation tasks. Just like the virtualization of storage, that is currently offered by the cloud, which has made the life easier for the organizations that have to keep stored content, the virtualization of rendering and software execution would be an important contribution to digital preservation, and significant relief for the responsible organizations. To realize this virtualization, and preserve the performability of these tasks as operating systems, protocols, format change, the provider of such services needs a repository and services like those presented in this paper. This could be done either by the community itself collaboratively, or provided (and charged) by the private sector.

[25] http://fedora-commons.org/

6.1 General Methodology for Applying the Dependency Management Approach

For applying the dependency management approach (independently of how) one can follow the next steps

1. Identify the desired tasks and objectives. This step strongly depends on the nature of the digital objects and the tasks that we want to perform on them. For instance if we suppose our domain is software, we can identify the following tasks: `Edit, Compile` and `Run`.
2. Model the identified tasks and their dependency types. If tasks can be hierarchically organized, then this should be done.
3. Specialize the Rule-based modeling according to the results of the previous step.
4. Capture the dependencies of the digital objects of the archive. This can be done manually, automatically or semi-automatically. Tools like PreScan [20] can aid this task. In addition this can be done in various levels of granularity: object-level (e.g. for a particular object), type-level (e.g. for all files of type html) and collection-level (e.g. for a collection of images).
5. Customize, use and exploit the dependency services according to the needs. For instance, task-performability services can be articulated with *monitoring* and *notification* services.
6. Evaluate the services in real tasks and curate accordingly the repository (return to Step 1).

6.1.1 Case Study: DANS

In the context of APARSEN we have also conducted a case study for the case of DANS (Data Archiving and Networked Services, NL)[26]. DANS aims at promoting sustained access to digital research data. For this purpose, it encourages researchers to archive and reuse data in a sustained manner, e.g. through the online archiving system EASY[27]. DANS also provides access, via NARCIS[28], to scientific datasets, e-publications and other research information in the Netherlands. Apart from these, the institute provides training and advice, and performs research into sustained access to digital information.

In collaboration with DANS, we have defined a number of scenarios that indicate where and how the dependency management approach could be used. The analysis yielded five main scenarios. For reasons of space here we present only their consolidation and two of them (for details see APARSEN D25.2).

In brief the desired (for DANS) tasks are related with the acceptable/preferred formats, and with the runability of DANS software (including computability of checksums).

Scenario 1
Description: For a number of data types (tables, text, images, etc.), specific file formats are considered as durable at least into the near future. DANS maintains a list of acceptable and preferred formats. These lists are the basis for file format migration activities. The list that DANS currently uses is publicly available[29].

Applicability: If the converters (or emulators) that are in use by DANS for carrying out the migration activities, are registered in a system like Epimenides, then the system can be exploited not only for checking whether a newly ingested file is in an acceptable/preferred format, but also for checking whether it is migratable to one preferred or acceptable format using the migration/emulation software that DANS uses and has registered.

Without such facility it is difficult for a curator to determine that (a) an archived dataset is formatted in a durable format and (b) to have an overview of the applicable file format migration procedures that can be carried out to convert a file into a preferred file format (given the fact that the list of preferred file formats will change over time as file formats might become obsolete).

Scenario 2
Description: As the usability and durability of file formats tend to change over time, for DANS it is important to periodically monitor and assess the applicability of the list of preferred formats and if it is necessary to replace a file format that became obsolete with a new one. Also new preferred formats can be introduced in the list. Specifically, say every year, the specifications on the list of preferred file formats have to be assessed based on a number of criteria (e.g. discussions in literature, consensus of organizations that provide guidelines in this field, etc.)

Applicability:
a. To add a new format in the list of acceptable/preferred file formats, the archivist can just register it to the knowledge base of `Epimenides`. The check performed at ingestion time will then function as expected (i.e.

[26] http://www.dans.knaw.nl/en
[27] http://easy.dans.knaw.nl
[28] http://www.narcis.nl
[29] Taken from `http://www.dans.knaw.nl/sites/default/files/file/EASY/DANSpreferredformatsUKDEF.pdf`

in accordance with the revised list of acceptable formats).

b. Before deleting a file format (or managing software) from the list of acceptable/preferred file formats (or available software respectively), the archivist can check the impact of that deletion, i.e. the impact that this deletion will have on the performability of tasks over the archived files. Recall the discussion on Section 4 about the Consequences of a Hypothetical Loss.
c. To delete a file format (or managing software) from the list of acceptable/preferred file formats (or available software respectively), the archivist can just delete the corresponding entries from the system. After doing so, the checking at ingestion time (Senario 1) will function as expected, i.e. in accordance with the revised list of acceptable formats.

Without such services it is difficult to identify all the consequences of file format's obsolescence. It is also difficult to identify what will happen if managing software that is able to convert to/from a preferred file format, is lost or will become obsolete.

6.1.2 Layering Tasks

We should stress that the modeling approach presented in this paper allows modeling and organizing tasks *hierarchically*. This is quite natural, and we have seen that the community and the literature many times attempts to provide a kind of layering. Below we describe, quite generally, some tasks. In some cases, the more we go down to the list, the more complex the tasks become, i.e. some of these tasks rely on the ability of performing other tasks.

Ability to:

- Retrieve the bits: Ability to get a particular set of stored bits.
- Access: Ability to retrieve the bits starting from an identifier (e.g. a persistent identifier)
- Render: Given a set of bits, ability to render them using the right symbol set (e.g. as defined in [9]) for creating the intended sensory impression.
- Run: Ability to run a program in a particular computer platform.
- Search: Ability to find a digital object. Search ability can be refined based on the type of the object (doc, structured, composite) and its searchable part (contents, structure, metadata).
- Link: Ability to place a digital object in context and exploit it. This may require combining data across difference sources.
- Assert Quality: Ability to answer questions of the form: What is its value of this digital object, is it authentic?
- Get Provenance: Ability to answer the corresponding questions (who, when, how).
- Assert Authenticity: (based on provenance, etc)
- Reproduce: Ability to reproduce a scientific result. This is crucial for e-Science.
- Update: Ability to update and evolve a digital object.
- Upgrade/Convert/Transform: Ability to upgrade a digital object (e.g. to a new format), or convert its form.

The approach presented in this paper is capable for modeling these tasks and their dependencies, as well as their hierarchical relationships.

6.2 Other Datasets

PRONOM[30] is an on-line information system about data file formats and their supporting software products. Recently *Linked Data PRONOM Lab*[31] emerged. It is based on Semantic Web technologies, specifically it is structured according to a draft vocabulary specification in RDF, and an RDF triplestore and a SPARQL Endpoint[32] are available. In comparison to the approach presented in this paper, we could say that PRONOM is less powerful since it does not model the notion of task, and does not support any kind of conversion-emulation reasoning. Nevertheless one could exploit some information from the PRONOM registry in order to enrich the knowledge base of `Epimenides`. However, it currently contains only 101 mime types, while the current knowledge base of `Epimenides` contains 647 different mime types, therefore the value of PRONOM is limited.

6.3 Quality of KB

This paper does not focus on collaborative use, although there is not any obstacle on using it collaboratively. In addition, crowdsourcing techniques, i.e. techniques allowing outsourcing a task to a large group of people in the form of an open call (as in [17,24]), can be adopted as well. In such cases, various versions and releases should be available (checked and closed, open, ongoing, etc), as well as peer evaluation, for controlling the evolution and quality over time. Such efforts can

[30] http://www.nationalarchives.gov.uk/PRONOM/Default.aspx
[31] `http://labs.nationalarchives.gov.uk/wordpress/index.php/2011/01/linked-data-and-pronom`
[32] `http://test.linkeddatapronom.nationalarchives.gov.uk/sparql/endpoint.php`

start with simple models (very broad tasks) and while the ontology/typology progresses, other methods (like those in [30]) can be used for assisting the identification of descriptions that lack specificity, and thus should be refined.

Finally we should mention that since the system uses the notion of profile, it allows hosting information which can be conflicting, so it does not require a global consensus.

7 Concluding Remarks

Each interoperability objective or challenge can be considered as a kind of demand for the performability of a particular task (or tasks), e.g. the task of exchanging data between two systems, the task of performing on the received data a certain operation, etc. However each task for being performed has various prerequisites (e.g. operating system, tools, software libraries, parameters, etc). We call all these dependencies.

The definition and adoption of standards (for data and services), aids interoperability because it is more probable to have (now and in the future) systems and tools that support these standards, than having systems and tools that support proprietary formats. From a dependency point of view, standardization essentially reduces the dependencies and makes them more easily resolvable; it does not vanish dependencies.

In all cases (standardization or not), we cannot achieve interoperability when the involved parties are not aware of the dependencies of the exchanged artifacts. However, the ultimate objective is the ability to perform a task, not the compliance to a standard. Even if a digital object is not compliant to a standard, there may be tools and processes that can enable the performance of a task on that object. As the scale and complexity of information assets and systems evolves towards overwhelming the capability of human archivists and curators (either system administrators, programmers and designers), it is important to aid this task, by offering services that can check whether it is feasible to perform a task over a digital object. For example, a series of conversions and emulations could make feasible the execution of software written in 1986 software on a 2013 platform. The process of checking whether this is feasible or not, could be too complex for a human and this is where advanced reasoning services, could contribute, because such services could greatly reduce the human effort required for periodically checking (monitoring) whether a task on a digital object is performable.

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

We proposed a methodology, we described the services which are more useful for the needs of digital preservation, and we showed how we can offer such services in an `RDF/S` implementation. In the sequel, we described a dataset and a prototype system, called `Epimenides`, that we have build based on the proposed approach which proves the technical feasibility, while the evaluation of its usability showed the underlying concepts can be easily understood by users. Although the knowledge base of the prototype system currently represents only some indicative tasks, it can demonstrate the benefits of the proposed approach.

From the technical side, an objective for future research is to develop 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, for an overview see [25]). Many thanks to Rene van Horik from DANS for his active participation, and to Anastasia Analyti for proofreading the paper.

References

1. D. Anderson, J. Delve, L. Konstantelos, A. Ciuffreda, and M. Dobreva. Totem: Trusted online technical environment metadata: a long-term solution for a relational database/rdf ontologies. 2011.
2. C. Becker, H. Kulovits, A. Rauber, and H. Hofman. Plato: a service oriented decision support system for preservation planning. In *Proceedings of the 8th ACM/IEEE-CS joint conference on Digital libraries*, pages 367–370. ACM, 2008.
3. C. Becker and A. Rauber. Decision criteria in digital preservation: What to measure and how. *JASIST*, 62(6):1009–1028, 2011.
4. F. Bellard. QEMU, a fast and portable dynamic translator. In *Procs of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
5. W. Bergmeyer. The KEEP Emulation Framework. In *Proceedings of the 1st International Workshop on Semantic Digital Archives (SDA 2011)*, 2011.
6. S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, I(1), 1989.

7. E. Conway, M. Dunckley, B. McIlwrath, and D. Giaretta. Preservation network models: Creating stable networks of information to ensure the long term use of scientific data. *In Proceedings of PV2009, Madrid, Spain*, pages 1–3, 2009.
8. E. Conway, B. Matthews, D. Giaretta, S. Lambert, M. Wilson, and N. Draper. Managing risks in the preservation of research data with preservation networks. *International Journal of Digital Curation*, 7(1):3–15, 2012.
9. M. Doerr and Y. Tzitzikas. Information Carriers and Identification of Information Objects: An Ontological Approach). 2012. CoRR, Digital Libraries, arXiv: 1201.0385v1 [cs.DL].
10. D. Giaretta (Editor). *Advanced Digital Preservation*. Springer, 2010.
11. D. Elenius, D. Martin, R. Ford, and G. Denker. Reasoning about Resources and Hierarchical Tasks Using OWL and SWRL. In *Proceedings of the 8th International Semantic Web Conference (ISWC'2009)*, 2009.
12. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. In *Proceedings of the 1st Conference on Social Semantic Web*, 2007.
13. S. Granger. Emulation as a digital preservation strategy. 2000. Corporation for National Research Initiatives.
14. S. Granger. Digital preservation & emulation: from theory to practice. *ICHIM (2)*, pages 289–296, 2001.
15. B. Haslhofer, E. Momeni Roochi, B. Schandl, and S. Zander. Europeana RDF store report. 2011.
16. 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. *W3C Member submission*, 21:79, 2004.
17. J. Howe. Crowdsourcing, why the power of the crowd is driving the future.
18. B. Lohman, B. Kiers, D. Michel, and van der J. Hoeven. Emulation as a Business Solution: the Emulation Framework. In *Proceedings of the 8th International Conference on Preservation of Digital Objects (iPres'2011)*, 2011.
19. R. A. Lorie. Long term preservation of digital information. In *Proceedings of the 1st ACM/IEEE-CS joint conference on Digital libraries*, JCDL '01, pages 346–352, New York, NY, USA, 2001. ACM.
20. Y. Marketakis, M. Tzanakis, and Y. Tzitzikas. PreScan: Towards Automating the Preservation of Digital Objects. In *Proceedings of the International Conference on Management of Emergent Digital Ecosystems MEDES'2009*, Lyon, France, October, 2009.
21. Y. Marketakis and Y. Tzitzikas. Dependency Management for Digital Preservation using Semantic Web technologies. *International Journal on Digital Libraries*, 10(4), 2009.
22. D. L. McGuinness and F. Van Harmelen. Owl web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.
23. K. Rechert, D. von Suchodoletz, and R. Welte. Emulation based services in digital preservation. In *Procs of the 10th annual joint conference on Digital libraries*, pages 365–368. ACM, 2010.
24. M. Sabou, K. Bontcheva, and A. Scharl. Crowdsourcing research opportunities: lessons from natural language processing. In *I-KNOW*, page 17, 2012.
25. A. Shaon, D. Giaretta, S. Crompton, E. Conway, B. Matthews, F. Marelli, U. Di Giammatteo, Y. Marketakis, Y. Tzitzikas, R. Guarino, H. Brocks, and F. Engel. Towards a Long-term Preservation Infrastructure for Earth Science Data. In *Proceedings of the 9th International Conference on Digital Preservation (iPres'2012)*, 2012.
26. C. Strubulis, Y. Tzitzikas, M. Doerr, and G. Flouris. Evolution of workflow provenance information in the presence of custom inference rules. In *3rd International Workshop on the role of Semantic Web in Provenance Management (SWPM12), co-located with ESWC12, Heraklion, Crete*, 2012.
27. M. Theodoridou, Y. Tzitzikas, M. Doerr, Y. Marketakis, and V. Melessanakis. Modeling and Querying Provenance by Extending CIDOC CRM. *J. Distributed and Parallel Databases (Special Issue: Provenance in Scientific Databases)*, 2010.
28. Y. Tzitzikas. "Dependency Management for the Preservation of Digital Information". In *Proceedings of the 18th Intern. Conf. on Database and Expert Systems Applications, DEXA'2007*, Regensburg, Germany, September 2007.
29. Y. Tzitzikas and G. Flouris. "Mind the (Intelligibily) Gap". In *Proceedings of the 11th European Conference on Research and Advanced Technology for Digital Libraries, ECDL'07*, Budapest, Hungary, September 2007. Springer-Verlag.
30. Y. Tzitzikas, M. Kampouraki, and A. Analyti. Curating the specificity of ontological descriptions under ontology evolution. *Journal on Data Semantics*, pages 1–32, 2013.
31. Y. Tzitzikas, Y. Marketakis, and G. Antoniou. Task-based Dependency Management for the Preservation of Digital Objects using Rules. In *Proceedings of the 6th Hellenic Conf. on Artificial Intelligence, SETN-2010*, Athens, Greece, 2010.
32. Y. Tzitzikas, Y. Marketakis, and Y. Kargakis. Conversion and Emulation-aware Dependency Reasoning for Curation Services . In *Proceedings of the 9th Annual International Conference on Digital Preservation (iPres2012)*, 2012.
33. J. Van der Hoeven, B. Lohman, and R. Verdegem. Emulation for digital preservation in practice: The results. *International Journal of Digital Curation*, 2(2), 2008.
34. J. R. Van Der Hoeven, R. J. Van Diessen, and K. Van Der Meer. Development of a universal virtual computer (uvc) for long-term preservation of digital objects. *Journal of Information Science*, 31(3):196–208, June 2005.
35. D. von Suchodoletz, K. Rechert, J. van der Hoeven, and J. Schroder. Seven steps for reliable emulation strategies–solved problems and open issues. In *Proceedings of the 7th Internationa Conference on Preservation of Digital Objects (iPRES'2010)*, pages 19–24, 2010.
36. D. Waters and J. Garrett. Preserving Digital Information Report of the Task Force on Archiving of Digital Information. In *Commissioned by the Commission on Preservation and Access and the Research Libraries Group, Inc., Washington DC: Commission on Preservation and Access.*, 1996.

Appendix

A Screen Dumps from `Epimenides`

Figure 14 shows the initial screen of the system, where the user can login for loading his personal or other profile. Figure 15 shows the first screen that allows the user to upload an atomic file or a zipped collection of files.

The system analyzes the contents of the zip file and for each of the included files it suggests a task. This is shown in Figure 16.

Figure 17 shows the results of this analysis. We can see that the first file is in red because the selected task, i.e. Rendenring, cannot be performed over that file digital object).In contrast, the selected tasks for the other two files can be performed, and for this reason they are marked with green.

The user can explore the dependencies for each one of the digital objects. For example Figure 18 shows what happens if the user clicks to explore the dependencies of the "Rendering" task.

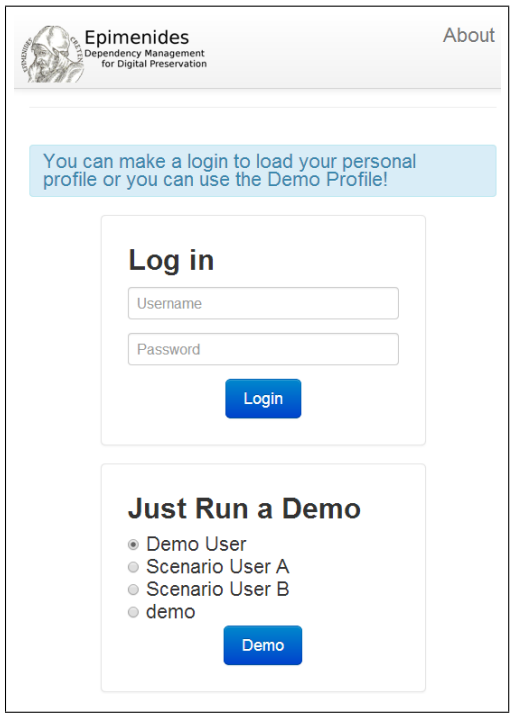

Fig. 14 Load your personal profile or use a demo profile

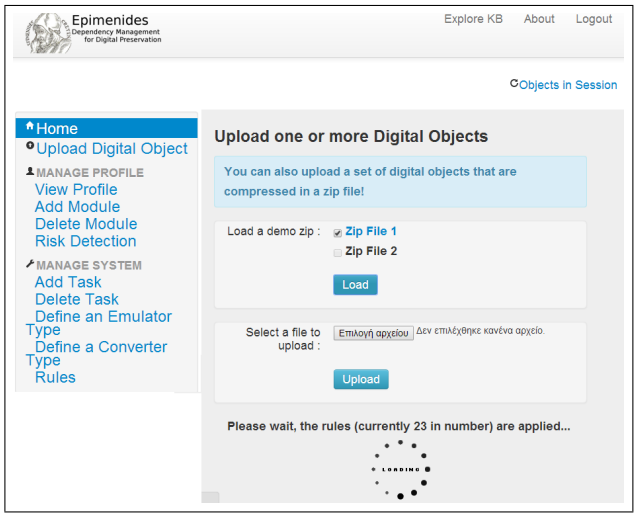

Fig. 15 Upload digital objects to check the performability of them

We can see all the rules of the selected task that are available in the system. The atoms of each rule are green or red. Green atoms are available in the profile of the user, while the red are not. Moreover the user can click on an atom to explore the dependencies of this atom, so he can see the rules or the facts of this atom.

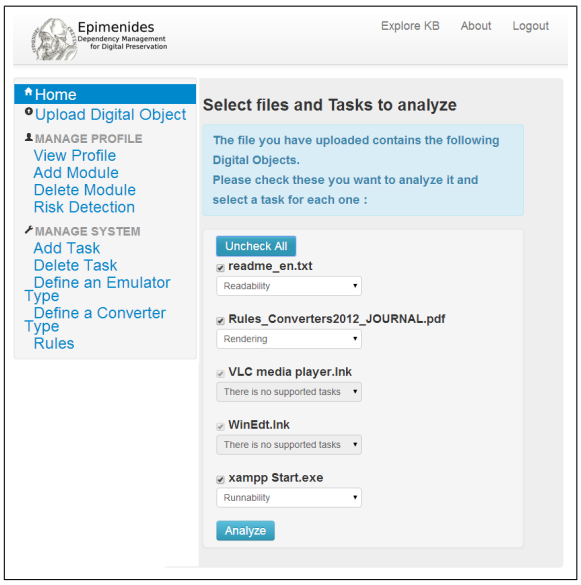

Fig. 16 System finds the tasks that usually make sense to apply to the uploaded digital objects

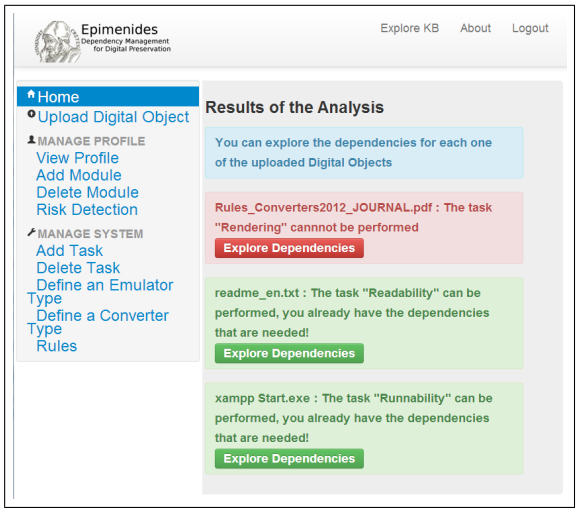

Fig. 17 Results of the analysis

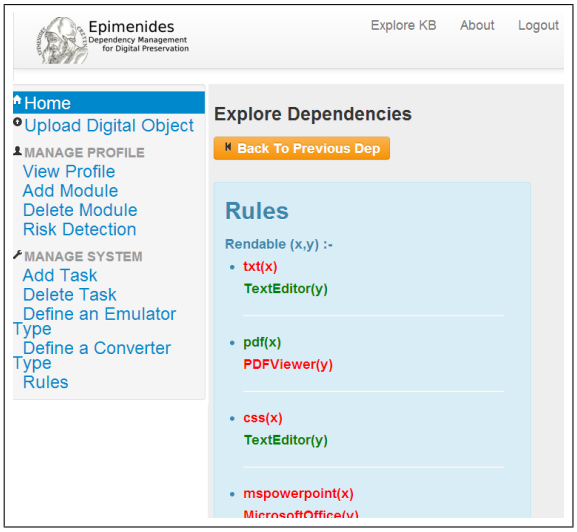

Fig. 18 Explore Dependencies of a Task