Task-based Dependency Management for the Preservation of Digital Objects using Rules

Yannis Tzitzikas, Yannis Marketakis, and Grigoris Antoniou

Computer Science Department, University of Crete,
Institute Of Computer Science, FORTH-ICS, GREECE
{tzitzik,marketak,antoniou}@ics.forth.gr

Abstract. The preservation of digital objects is a topic of prominent importance for archives and digital libraries. This paper focuses on the problem of preserving the *performability* of tasks on digital objects. It formalizes the problem in terms of Horn Rules and details the required inference services. The proposed framework and methodology is more expressive and flexible than previous attempts as it allows expressing the various properties of dependencies (e.g. transitivity, symmetry) straightforwardly. Finally, the paper describes how the proposed approach can be implemented using various technologies.

1 Introduction

The preservation of digital objects is a topic of prominent importance for archives and digital libraries. To support digital preservation, i.e. the curation of archives of digital objects, requires tackling several issues. For instance, there is a need for services that help archivists in checking whether the archived digital artifacts remain *functional*, in identifying *hazards* and the consequences of probable losses or *obsolescence risks*.

To tackle these requirements [11] showed how the needed services can be reduced to dependency management services, while [12] extended that model with disjunctive dependencies. The key notions of these works is the notion of *module*, *dependency* and *profile*. In a nutshell, a *module* can be a software/hardware component or even a knowledge base expressed either formally or informally, explicitly or tacitly, that we want to preserve. A module may require the availability of other modules in order to function, be understood or managed. We can denote such *dependency relationships* as $t > t'$ meaning that module t depends on module t'. A *profile* is the set of modules that are assumed to be known (available or intelligible) by a user (or community of users). Based on this model, a number of services have been defined for checking whether a module is intelligible by a community, or for computing the *intelligibility gap* of a module. `GapMgr`[1] and `PreScan`[2] [9] are two systems that have been developed based on this model, and have been applied successfully in the context of the EU project CASPAR[3].

[1] http://athena.ics.forth.gr:9090/Applications/GapManager/
[2] http://www.ics.forth.gr/prescan
[3] http://www.casparpreserves.eu/

In the current work we extend that framework with *task*-based dependencies. This extension allows expressing dependencies in a more systematic manner, i.e. each dependency is due to one or more tasks. We found the extended framework on Horn Rules and we sketch a methodology for applying it. The proposed framework and methodology, apart from simplifying the disjunctive dependencies of [12], is more expressive and flexible as it allows expressing the various properties of dependencies (e.g. transitivity, symmetry) straightforwardly.

The rest of this paper is organized as follows. Section 2 introduces a running example and discusses requirements. Section 3 contains background information on Datalog. Section 4 introduces the proposed approach. Section 5 elaborates on the inference services required for task-performability, risk-detection and computing intelligibility gaps, and Section 6 discusses implementation choices. Section 7 discusses related work, and finally, Section 8 concludes and identifies issues that are worth further research.

2 Motivation and Requirements

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

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

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

1. NotePad is a TextEditor
2. VI is a TextEditor
3. HelloWorld.java is a JavaSourceFile
4. HelloWorld.cc is a C++SourceFile
5. javac1.6 is a JavaCompiler
6. JRE1.5 is a JavaVirtualMachine
7. A file is Editable if it is a TextFile and a TextEditor is available

8. A file is JavaCombilable if it is a JavaSourceFile and a JavaCompiler is available
9. A file is C++Combilable if it is a C++SourceFile and a C++Compiler is available
10. A file is Compilable if it is JavaCompilable or C++Compilable
11. If a file is Editable then it is Readable

Lines 1-6 are actually facts while lines 7-11 define how various tasks are carried out. Notice that some facts are valid for James while some other are valid for Helen (the only fact that is not valid for James is 2, while for Helen only 2 and 6 hold). From these we can infer that James is able to compile the file `HelloWorld.java` (using the lines 3,5,8,10) but he cannot compile the file `HelloWorld.cc` (since there is no fact about C++Compiler for James). If James send his TextFiles to Helen then she can only edit them but not compile them since she has no facts about Compilers.

Requirements In general, we have identified the following key requirements:

- **Task-Performability Checking.** In most cases, to perform a task we have to perform other subtasks and to fulfil associated requirements for carrying out these tasks (e.g. to have the necessary modules - in our running example the necessary digital files). 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.
- **Risk Detection.** The 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 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**. It is desirable to be able to define task-type hierarchies for gaining flexibility and reducing the number of rules that have to be defined.
- **Properties of Dependencies.** Some dependencies are transitive, some are not. Therefore we should be able to define the properties of each kind of dependency.

3 Background: Datalog

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

Syntax The basic elements of Datalog are: *variables* (denoted by a capital letter), *constants* (numbers or alphanumeric strings), and *predicates* (alphanumeric strings). A *term* is either a constant or a variable. A constant is called *ground term* and the *Herbrand Universe* of a Datalog program is the set of constants occurring in it. An *atom* $p(t_1, ..., t_n)$ consists of an n-ary predicate symbol p and a list of arguments $(t_1, ..., t_n)$ such that each t_i is a term. A *literal* is an atom $p(t_1, ..., t_n)$ or a negated atom $\neg p(t_1, ..., t_n)$. A *clause* is a finite list of literals, and a *ground clause* is a clause which does not contain any variables. Clauses containing only negative literals are called *negative clauses*, while *positive clauses* are those with only positive literals in it. A *unit clause* is a clause with only one literal. *Horn Clauses* contain at most one positive literal. There are three possible types of Horn clauses, for which additional restrictions apply in Datalog:

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

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. Furthermore, there are built-in predicates like e.g. $=, \neq, <$, which we do not discuss explicitly here.

If S is a set of positive unit clauses, then $E(S)$ denotes the extensional part of S, i.e. the set of all unit clauses in S whose predicates are elements of $EPred$. On the other hand, $I(S) = S - E(S)$ denotes the intensional part of S (clauses in S with at least one predicate from $IPred$). Now we can define a *Datalog program* P as a finite set of Horn clauses such that for all $C \in P$, either $C \in EPred$ or C is a safe rule where the predicate occurring in the head of C belongs to $IPred$.

So far, we have described the syntax of pure Datalog. In order to allow also for negation, we consider an extension called *stratified Datalog*. Here negated literals in rule bodies are allowed, but with the restriction that the program must be *stratified*. For checking this property, the *dependency graph* of a Datalog program P has to be constructed. For each rule in P, there is an arc from each predicate occuring in the rule body to the head predicate. P is stratified iff whenever there is a rule with head predicate p and a negated subgoal with predicate q, then there is no path in the dependency graph from p to q.

Semantics The *Herbrand base* (HB) of a Datalog program is the set of all possible ground unit clauses that can be formed with the predicate symbols and the constants occurring in the program. Furthermore, let EHB denote the extensional and IHB the intensional part of HB. An *extensional database* (EBD)

is a subset of EHB, i.e. a finite set of positive ground facts. In deterministic Datalog, a Herbrand interpretation is a subset of the Herbrand base HB. For pure Datalog, there is a least Herbrand model such that any other Herbrand model is a superset of this model. Stratified Datalog is based on a closed-world assumption. If we have rules with negation, then there is no least Herbrand model, but possibly several minimal Herbrand models, i.e. there exists no other Herbrand model which is a proper subset of a minimal model. Among the different minimal models, the one chosen is constructed in the following way: When evaluating a rule with one or more negative literals in the body, first the set of all answer-facts to the predicates which occur negatively in the rule body is computed (in case of EDB predicates these answer-facts are already given), followed by the computation of the answers to the head predicates. For stratified Datalog programs, this procedure yields a unique minimal model. The minimum model computed in this way is often called the *perfect Herbrand model*.

4 Proposed Approach

In brief, digital files are represented by EDB facts. Task-based dependencies (and their properties) are represented as Datalog rules and facts. Profiles (as well as particular software archives or system settings) are represented by EDB facts. Datalog query answering and methods of logical inference (i.e. deductive and abductive reasoning) are exploited for enabling the required inference services (performability, risk detection, etc).

4.1 Digital Files and Type Hierarchies

Digital files and their types are represented as EDB facts. The two files of our running example will be expressed as:

```
JavaFile(HelloWorld.java).
C++File(HelloWorld.cc).
```

The types of the digital files can be organized hierarchically. Such taxonomies can be represented with appropriate rules. For example to define that every `JavaFile` is also a `UTF8File` we must add the following rule `UTF8File(X) :- JavaFile(X)`. 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. `ReadOnly(HelloWorld.java)`, `LastModifDate(HelloWorld.java, 2008-10-18)`. Also note that in place of the filenames, we could use any string that can be used as an identity of these files (e.g. file paths, URIs, DOIs, etc).

4.2 Software Components

Software components can be described analogously, e.g.:

```
AsciiEditor(NotePad).        |  C++Compiler(gcc).
AsciiEditor(vi).             |  JVM(jre1.5win).
JavaCompiler(javac 1.6).     |  JVM(jre1.6linux).
```

Again predicates are used for expressing the types of the software components. The above set of facts may correspond to the software components available in a particular computer.

4.3 Task-Dependencies

We will also use (IDB) predicates to model tasks and their dependencies. Specifically, for each real world task we define two intensional predicates: one (which is usually unary) to denote the task, and another one (with arity greater than 2) for denoting the dependencies the task. For instance, `Compile(HelloWorld.java)` will denote the task of compiling `HelloWorld.java`. Since the compilability of `HelloWorld.java` depends on the availability of a compiler (specifically a compiler for the Java language), we can express this dependency using a rule of the form: `Compile(X) :- Compilable(X,Y)` where the binary predicate `Compilable`(X, Y) is used for expressing the appropriateness of a Y for compiling a X. For example, `Compilable(HelloWorld.java, javac 1.6)` expresses that `HelloWorld.java` is compilable by `javac 1.6`. It is beneficial to express such relationships at the class level (not at the level of individuals), specifically over the types (and other properties) of the digital objects and software components, i.e. with rules of the form:

```
Compilable(X,Y) :- JavaFile(X), JavaCompiler(Y).
Compilable(X,Y) :- C++File(X), C++Compiler(Y).
Runable(X,Y)    :- JavaClassFile(X), JVM(Y).
EditableBy(X,Y) :- JavaFile(X), AsciiEditor(Y).
```

Relations of higher arity can also be employed according to the requirements, e.g.:

```
Runnable(X) :- Runnable(X,Y,Z)
Runnable(X,Y,Z) :- JavaFile(X), Compilable(X,Y), JVM(Z)
```

4.4 Task Hierarchies

We have already seen how file type hierarchies can be expressed using rules. We can express hierarchies of tasks in a similar manner. The motivation is the need for enabling deductions of the form: "if we can do task A then certainly we can do task B". For example:

```
Read(X) :- Edit(X).
Read(X) :- Compile(X).
```

The first rule means that if we can edit something then certainly we can read it too. Alternatively, or complementarily, we can define such deductions at the "dependency" level, e.g.:

```
Readable(X,Y) :- EditableBy(X,Y).
Intelligible(X,Y) :- ReadableBy(X,Y).
```

4.5 Properties of (Task) Dependencies

We can also express other properties of task dependencies (e.g. transitivity, symmetry, etc). 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 [11] (i.e. the transitivity of dependencies).

4.6 Profiles

A profile is a set of facts, describing the modules available (or assumed to be known) to a user (or community). For example, the profiles of James and Helen can be expressed as two sets of facts:

```
James                        |    Helen
-----------------------------+------------
AsciiEditor(NotePad).        |    AsciiEditor(Vi).
JavaCompiler(javac 1.6).     |    JVM(jre1.5Win).
JVM(jre1.5Win).              |
```

Synopsis Methodologically, for each real world task we define two intensional predicates: one (which is usually unary) to denote the task, and another one (which is usually binary) for denoting the dependencies of task (e.g. `Read, Readable`). Figure 1 depicts the partitioning of the various facts and rules. For instance, all services regarding James should be based on James'box plus the boxes at the upper level.

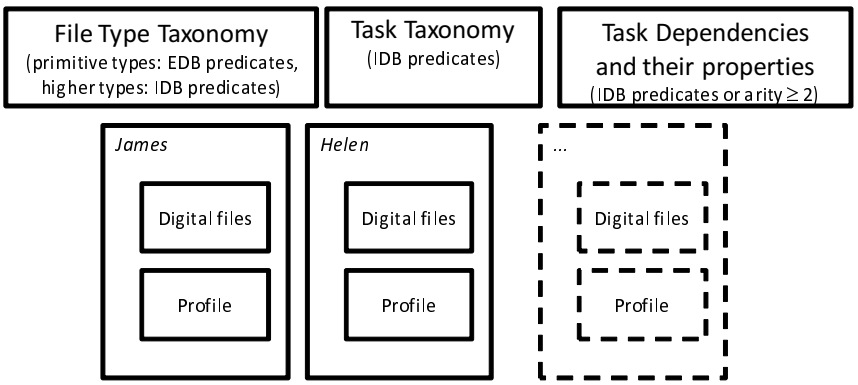

Fig. 1. Logical partitioning

5 Reasoning Services

Here we describe how the reasoning services described at Section 2 can be realized in the proposed framework.

- **Task-Performability**. This service aims at answering if a task can be performed by a user/system. It relies on query answering over the Profiles of the user. E.g. to check if `HelloWorld.cc` is compilable we have to check if `HelloWorld.cc` is in the answer of the query `Compile(X)`.
- **Risk-Detection**. Suppose that we want to identify the consequences on *editability* after removing a module, say `NotePad`. To do so we can do the following:
  1. We compute the answer of the query `Edit(X)`, and let A be the returned set of elements.
  2. We delete `NotePad` from the database and we do the same. Let B be the returned set of elements[4].
  3. The elements in $A \setminus B$ are those that will be affected.
- **Computation of Gaps (Missing Modules)**. There can be more than one way to fill a gap due to the disjunctive nature of dependencies since the same predicate can be the head of more than one rules (e.g. the predicate `AsciiEditor` in the example earlier). The gap is actually the set of facts that are missing and are needed to perform a task. To this end we must find the possible explanations (the possible facts) that entail a consequence (in our case a task). For example some possible explanations for the compilability of a JavaFile is the existence of the available compilers. In order to find the possible explanations of a consequence we can use abduction [8, 4, 5]. Abductive reasoning allows inferring an atom as an explanation of a given consequence. For example assume that the file `HelloWorld.cc` is not compilable. Abduction will result all the possible C++ Compilers as explanations for the compilability of the file.

6 Implementation and Application Issues

There are several possible implementation approaches. Below we describe them in brief:

Prolog is a declarative logic programming language, where a program is a set of Horn clauses describing the data and the relations between them. The proposed approach can be straightforwardly expressed in Prolog. Furthermore and regarding abduction there are several approaches that either extend Prolog [2] or augment it [3] and propose a new Programming Language.

The **Semantic Web Rule Language (SWRL)** [7] is a combination of OWL DL and OWL Lite [10] with the Unary/Binary Datalog RuleML[5]. SWRL

[4] In Prolog we could use the *retract* feature.
[5] http://ruleml.org

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

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

Just indicatively, Table 1 synopsizes the various implementation approaches.

Table 1. Implementation Approaches

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

7 Related Work

There are many dependency management related works but only a few focus on task-based dependencies. Below we discuss in brief some of these works. [1] proposes a static deployment system for ensuring the success of two tasks: *installation* and *deinstallation*. It is based on a dependency description language, where the requirements of a service are expressed in first order predicate language in conjunctive normal form. The success of installation guarantees that once a component is installed successfully it will work properly while the success of deinstallation ensures that the system remains safe after the removal of a component. [6] defines four types of dependencies: *goal*, *soft goal*, *task* and *resource* dependencies. The first three types determine the conditions or the particular ways under which a specific goal or task can be attained. Furthermore it describes several properties, for soft goal dependencies, that determine the best

approach to be followed. Finally it categorizes components to light (replaceable components) and heavy (components on which others strictly depend). In brief, we can say that all these approaches are less flexible and extensible (in terms of task and dependency modeling) than the approach that we propose.

8 Concluding Remarks

We showed how rules can be employed for advancing the dependency management services that have been proposed for digital preservation. We reduced the problem to Datalog-based modeling and query answering. One issue that is worth further research is to investigate whether the way abduction is supported by existing systems (e.g. [2, 3]) is adequate for the problem at hand.

Other issues that are important for applying this model successfully in real settings is how *modularity* is supported and what kind of assisting tools are needed for managing and administrating the underlying sets of facts and rules.

References

1. M. Belguidoum and F. Dagnat. Dependency Management in Software Component Deployment. *Electronic Notes in Theoretical Computer Science*, 182:17–32, 2007.
2. H. Christiansen and V. Dahl. Assumptions and abduction in Prolog. In *3rd International Workshop on Multiparadigm Constraint Programming Languages, MultiCPL*, volume 4. Citeseer, 2004.
3. H. Christiansen and V. Dahl. HYPROLOG: A new logic programming language with assumptions and abduction. *Lecture Notes in Computer Science*, 3668:159–173, 2005.
4. L. Console, D.T. Dupre, and P. Torasso. On the relationship between abduction and deduction. *Journal of Logic and Computation*, 1(5):661, 1991.
5. T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM (JACM)*, 42(1):3–42, 1995.
6. X. Franch and N.A.M. Maiden. Modeling Component Dependencies to Inform their Selection. *2nd International Conference on COTS-Based Software Systems, Springer*, 2003.
7. Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. "SWRL: A Semantic Web Rule Language Combining OWL and RuleML", May 2004. (http://www.w3.org/Submission/SWRL/).
8. AC Kakas, RA Kowalski, and F. Toni. The Role of Abduction in Logic Programming. *Handbook of Logic in Artificial Intelligence and Logic Programming: Logic programming*, page 235, 1998.
9. Y. Marketakis, M. Tzanakis, and Y. Tzitzikas. PreScan: Towards Automating the Preservation of Digital Objects. In *Procs of the Intern. Conf. on Management of Emergent Digital Ecosystems MEDES'2009*, Lyon, France, October, 2009.
10. D. L. McGuinness and F. van Harmelen. "OWL Web Ontology Language Overview", 2004. (http://www.w3.org/TR/owl-features/).
11. Y. Tzitzikas. "Dependency Management for the Preservation of Digital Information". In *Procs of the 18th International Conference on Database and Expert Systems Applications, DEXA'2007*, Regensburg, Germany, September 2007.
12. Y. Tzitzikas and G. Flouris. "Mind the (Intelligibily) Gap". In *Procs of the 11th European Conference on Research and Advanced Technology for Digital Libraries, ECDL'07*, Budapest, Hungary, September 2007. Springer-Verlag.