Mind the (Intelligibility) Gap

Yannis Tzitzikas[1,2] and Giorgos Flouris[3]

[1] Computer Science Department, University of Crete, GREECE
[2] Institute of Computer Science, FORTH, GREECE
[3] Istituto della Scienza e delle Tecnologie della Informazione, CNR, ITALY
tzitzik@ics.forth.gr, flouris@isti.cnr.it

Abstract. Intelligibility, evolution and emulation are some of the key notions for digital information preservation. In this paper we define formally these notions on the basis of modules and inter-module dependencies. Subsequently, we discuss how we can handle the evolution of modules and dependencies. This work can be exploited for building advanced preservation information systems and registries.

1 Introduction

Modern society and economy is increasingly dependent on a deluge of only digitally available information. The preservation of digital information within an unstable and rapidly evolving computing environment is a challenging problem of prominent importance. [8] proposed tackling this problem on the basis of the notion of *dependency*. In this paper we adopt the same abstract notion of *module* and *dependency* but in a more expressive framework. We formalize the concept of *intelligibility* of digital objects and we extend the model with *emulators*. As preservation is an endless process, we also focus on the evolution of dependencies and describe change operations and notification services. This work can be exploited for building advanced preservation information systems and registries.

2 Dependencies and Dependency Graphs

2.1 Definitions and Notations

An archive's digital collection consists of a set of *objects*, containing all the data objects in the archive, as well as of a set of *modules* (or components) needed for understanding/executing/managing such objects. In this paper, we adopt a very general interpretation of the term module. It can be a software or hardware module; it could also be a knowledge model expressed either formally or informally, explicitly or tacitly; it could even be a digital object describing how another module functions (*e.g.*, a manual). For instance, it could be an English-To-Greek dictionary that is useful for a Greek-speaking person to understand a piece of text written in English, or an ontology that is useful for understanding the contents of a metadata file. Thus, the distinction between digital objects

and modules is often vague, so, we will keep our terminology simple by using the term module to refer to both modules and objects.

There is no standard method for defining what a module is, as we may have modules of various levels of abstraction. An element modeled as one module could in fact correspond to a large number of interconnected finer modules, depending on the level of detail that we are interested at and/or is useful for the application at hand. Similarly, a software module replicated in several places can be viewed as a single module, or as a compound module consisting of a number of replicas, either of which needs to be intelligible. Finally, complex modules, *e.g.*, a web page, may consist of images, text etc, and could be viewed as a single module, or as many, all of which should be intelligible.

Modules (or objects) may require the availability of one or more other modules in order to function (or be understood). We can model this using a *dependency relation*, denoted by $>$, where $t > t'$ means that t depends on t', *e.g.*, it may mean that t cannot function without the existence of t'. In principle, t' also depends on some other module an so on, and such dependencies may continue indefinitely, as probably nothing in this world is self-existent. Nevertheless, depending on the application, we may consider some modules to be understandable by all users of a digital archive; such modules will be called *primitive*.

Modules, in general, do not depend on one module but many. Consider for example a file `README.TXT` written in English; the intelligibility of the file depends on the availability of a suitable text editor (*e.g.*, Notepad), plus a good understanding of the English language by the reader. This can be modeled using two dependencies of the form $t_{README} > t_{NOTEPAD}$, $t_{README} > t_{ENG}$. This pair of dependencies has conjunctive semantics, in the sense that t_{README} requires both $t_{NOTEPAD}$ and t_{ENG} in order to be understood.

In other cases, dependencies could have disjunctive semantics; for example, the above file can be read using, *e.g.*, Wordpad, even if Notepad is not available. To capture this kind of semantics, we will generalize our notations, by defining the concept of the *generalized module*, which is just a set of modules (*e.g.*, $\{t_1, t_2\}$). A generalized module is interpreted disjunctively, in the sense that $\{t_1, t_2\}$ means "either t_1, or t_2". Standard modules can be captured using singleton sets, *e.g.*, $\{t\}$.

Generalizing our notations, we will henceforth use $>$ to denote the dependency relation between generalized modules. This way, the dependency $\{t_{README}\} > \{t_{NOTEPAD}, t_{WORDPAD}\}$ means that the intelligibility of the module t_{README} depends on the availability of at least one of $t_{NOTEPAD}$, $t_{WORDPAD}$.

Thus, there are two basic dependency types. The first is conjunctive dependencies, which are useful when there are some modules (*e.g.*, $t_1, t_2, \ldots$) which are all necessary for the intelligibility of a module t; this type is modeled using a number of different dependencies, *i.e.*, $\{t\} > \{t_1\}$, $\{t\} > \{t_2\}$, ... The second is disjunctive ones, used when t requires the existence of at least one of $t_1, t_2, \ldots$ for its intelligibility; this type is captured using generalized modules, *i.e.*, $\{t\} > \{t_1, t_2, \ldots\}$.

The above basic types allow us to model many different types of dependencies, including quite complex ones. For example, if we want to model that "the readability of t_{README} depends on the existence of t_{ENG}, and either $t_{NOTEPAD}$ or $t_{WORDPAD}$", we can capture it using the pair of dependencies $\{t_{README}\} > \{t_{ENG}\}$ and $\{t_{README}\} > \{t_{NOTEPAD}, t_{WORDPAD}\}$. A more difficult case is if we want to model that "the readability of t_{README} depends on either knowledge of English (t_{ENG}), or knowledge of Greek (t_{GR}) and an English-to-Greek dictionary (t_{ENG2GR})"; this would require the extra step of transforming this description into the equivalent one: "t_{README} depends on either t_{ENG} or t_{GR} and either t_{ENG} or t_{ENG2GR}", which can be captured using the pair: $\{t_{README}\} > \{t_{ENG}, t_{GR}\}$ and $\{t_{README}\} > \{t_{ENG}, t_{ENG2GR}\}$[1].

We now have all the necessary ingredients for the formal definition of our model. We denote by $\mathcal{T}$ the set of all *modules* (which include digital objects as well); a *generalized module*, also called a *node*, is any set S of modules ($S \subseteq \mathcal{T}$), interpreted disjunctively. Thus, the set of all generalized modules is just the powerset of $\mathcal{T}$, denoted by $2^{\mathcal{T}}$. A *dependency relation* is a binary relation $> \subseteq 2^{\mathcal{T}} \times 2^{\mathcal{T}}$; the notation $S_1 > S_2$ implies that at least one module of S_1 depends on at least one module of S_2. These notions can be more intuitively represented in a graph $\Gamma = (2^{\mathcal{T}}, >)$, which we will call the *dependency graph*. Sometimes, it will be useful to refer to *families of nodes*, which are conjunctively interpreted sets of generalized modules, denoted by $\mathbf{S}$; notice that $\mathbf{S} \subseteq 2^{\mathcal{T}}$, *i.e.*, each element of $\mathbf{S}$ is a generalized module S (*i.e.*, a set of modules, interpreted disjunctively).

As explained before, certain notions, like module, primitive module, dependency etc are just application-dependent conventions. In the following, we assume that a dependency graph contains (models) all the modules and their dependency-related information that is important for the application at hand. Moreover, we make no assumptions as to the properties of $>$ (*e.g.*, acyclic, transitive etc), as such assumptions may be invalid for certain applications.

2.2 Types of Dependencies

Consider a relationship $S_1 > S_2$. We can distinguish the following general cases, depending on the size of S_1:

- $|S_1| = 1$, $|S_2| \geq 1$ (*e.g.*, $S_1 = \{t_1\}, S_2 = \{t_{21}, t_{22}, \ldots, t_{2m}\}$). Here, $S_1 > S_2$ means that t_1 depends on one of $t_{21}, t_{22}, \ldots, t_{2m}$. Such dependencies will be called *basic*.
- $|S_1| > 1$, $|S_2| \geq 1$ (*e.g.*, $S_1 = \{t_{11}, t_{12}, \ldots, t_{1n}\}$, $S_2 = \{t_{21}, t_{22}, \ldots, t_{2m}\}$). Here, $S_1 > S_2$ means that one of $t_{11}, t_{12}, \ldots, t_{1n}$ depends on one of $t_{21}, t_{22}, \ldots, t_{2m}$. Such dependencies will be called *complex*.

The above distinction is motivated by our belief that complex dependencies are artificial and probably not useful in practice. For example, the (complex) dependency $\{t_1, t_2\} > \{t_3, t_4\}$ implies that either t_1 or t_2 depend on either t_3 or t_4. Most often, this will be just because, for example, t_1 (alone) depends on

[1] This idea is based on the algorithm transforming logical formulas in CNF.

t_3, and t_2 (alone) depends on t_4. It is hard to find an example where a complex dependency is not just a consequence of a number of basic ones. Another type of dependencies is *trivial* dependencies. A dependency $S_1 > S_2$ is called trivial iff $S_1 \subset S_2$. Trivial dependencies can be either basic or complex and they are always true. For example, $\{t_1, t_2\}$ always depends on $\{t_1, t_2, t_3\}$ because, if neither of t_1, t_2, t_3 is understandable, then, obviously, none of t_1, t_2 is understandable either. Trivial dependencies could be considered as the counterpart of tautologies in a logical theory. Finally, dependencies where either $S_1 = \emptyset$ or $S_2 = \emptyset$ are not intuitively useful, as they have no real-world counterpart.

The above observations imply that the only interesting dependencies are those that are both basic and non-trivial; such dependencies will be called *editable*. In the rest of this paper, the symbol $>$ will always refer to editable relations, and a dependency graph will be assumed to contain only editable arcs.

3 Intelligibility

3.1 Profiles

Now, let us consider a preservation system, say s, which supports a finite number of users, say $u_1, \ldots, u_n$, by archiving the digital objects that may be of interest to them, as well as the modules that are useful for their intelligibility. The related information (dependencies), that is useful for the system to determine the useful modules is modeled in a dependency graph $\Gamma = (2^{\mathcal{T}}, >)$. This graph Γ is assumed to capture the (known and interesting) state of affairs regarding the dependencies between the various models and objects available in the "world", and may contain information on modules that are not available to the system and/or any of its users.

The system s, as well as any of the users $u_1, \ldots, u_n$, are assumed to have access to some of the modules in $\mathcal{T}$; the sets of modules that they have access to are called (system or user) *profiles* and denoted by T_s and T_{u_i} respectively. Notice that only atomic modules are included in a profile, as it makes no intuitive sense to say that someone has access to either module t_1 or module t_2.

3.2 Modules' Intelligibility, Self-Intelligibility, Intelligibility Gaps

It is often useful to be able to determine whether a user u, with a profile T_u, can understand a module $t \in \mathcal{T}$. Using our definitions, in order for t to be understood, the modules that it depends upon should be available. To capture this notion, we define the family of nodes that are *directly required for intelligibility*, denoted by $req(t)$, as follows: $req(t) = \{S \subseteq \mathcal{T} \mid \{t\} > S\}$. Given the disjunctive nature of nodes, in order for t to be understood by u, he must have access to at least one module from each node $S \in req(t)$; that is: $S \cap T_u \neq \emptyset$ for all $S \in req(t)$.

But this is not enough, because the modules required for understanding t, should themselves be intelligible (not just accessible) by u. Normally, we can assume that this is true; if a user can access some module t', he has probably

taken actions already so as to make t' intelligible, by importing all the necessary modules, thus making his profile *self-intelligible*: T_u is self-intelligible, iff for all $t \in T_u$ and for all $S \in req(t)$ it holds that $S \cap T_u \neq \emptyset$.

Using the notion of self-intelligibility, our original question on the intelligibility of a module t can be answered as follows: if the user's profile is self-intelligible, all we need to check is whether $S \cap T_u \neq \emptyset$ for all $S \in req(t)$. It can be easily shown that this will be true iff the set $T_u \cup \{t\}$ is self-intelligible. In the general case, we can say that a module t is intelligible by a user u iff there is some profile ${T_u}' \subseteq T_u$ such that ${T_u}' \cup \{t\}$ is self-intelligible.

A module t not being understandable by u, means that there are certain "missing" modules, which, if added to T_u, will make t intelligible; such modules are denoted by $Missing(t, u)$ and form an *intelligibility gap*. Similarly, we can define $Missing(t, s)$ for the system.

3.3 Algorithmic Aspects

Let's now see how the above quantities can be determined algorithmically. Firstly, the algorithm for determining self-intelligibility is trivial and follows from the definition. Moreover, if T_u is self-intelligible, then, determining whether t is intelligible by u is equivalent to determining whether $T_u \cup \{t\}$ is self-intelligible.

Unfortunately, this technique cannot be applied for a non-self-intelligible profile T_u, because we would have to check the self-intelligibility of ${T_u}' \cup \{t\}$ for all ${T_u}' \subseteq T_u$; this is not an efficient calculation. To address this problem we reduce it to the problem of query answering in monadic and negation-free Datalog. For reasons of space below we just sketch this reduction. If S is a node, we denote by $S^\vee$ the disjunction of all modules in S, *i.e.*, $S^\vee = \vee_{t \in S} t$. If $t \in \mathcal{T}$, then for each S such that $t > S$ we can define $S^\vee$ and then take the conjunction of these disjunctions, *i.e.*, for each t we define the logical formula $E^{CNF}(t) = \wedge_{t>S} S^\vee$. Let $E^{DNF}(t)$ be the equivalent logical formula in DNF. For each conjunction, say $t_1 \wedge t_2$ in $E^{DNF}(t)$ we derive the Datalog rule $t(X) : -t_1(X), t_2(X)$. Let $R(t)$ denote the resulting set of rules and $R(\Gamma)$ the union of the rules for each module in Γ. Now for each $t \in T_u$ that is *primitive*, we derive the fact $t(Const)$ where $Const$ is a constant, and let $R(T_u)$ denote these facts. We use the same constant $Const$ for each $t \in T_u$. It can be easily proved that if the answer of the query $q = t$ in the knowledge base $R(\Gamma) \cup R(T_u)$ is not empty, specifically if it equals $\{Const\}$, then t is intelligible by u. Otherwise, it is not intelligible, so it belongs to the gap. Figure 1, illustrates the reduction with an example. Notice that the profile $T_u = \{t_1, t_2, t_4\}$ is not self-intelligible because t_1 is not intelligible since $t_1 > \{t_5, t_6\}$ and $\{t_5, t_6\} \cap T_u = \emptyset$. In this example t is not intelligible by u, however t_8 is intelligible by u. Indeed the answer of the query $q = t$ is empty, while the answer of the query $q = t_8$ is not empty[2].

As Γ is expected to change less frequently than the profiles, we can keep the rules $R(\Gamma)$ stored (to avoid recomputing them); on the other hand, recomputing $R(T_u)$ is faster, so frequent changes in the profiles should not cause major delays.

[2] Notice that if T_u were self-intelligible, then t would be intelligible simply because for each $S \in req(t)$, we have $S \cap T_u \neq \emptyset$.

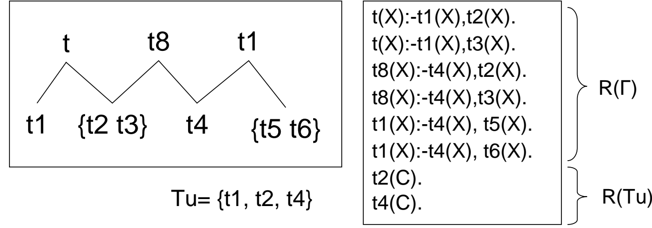

Fig. 1. Deciding intelligibility using Datalog

Computing $Missing(t, u)$ is more difficult, as, due to the disjunctive nature of dependencies, there may more than one possible solution. Thus, different criteria of minimality (*e.g.*, cardinality) could be adopted to select a single solution. This problem is quite relevant with abduction [2] and is a subject for future research.

4 Emulators

When a module, say t, requires (depends on) a particular module, say t_1, which is not available, we may consider using some kind of emulator, say t_2, which would make t dependent on another, available module, say t_3. In this case, we say that t_1 (the *emulated module*) was *emulated* by t_2 (the *emulator*) and t_3 (the *emulation target*); the emulator and the emulation target will be collectively referred to as the *emulation scheme* and the whole process will be called *emulation*. Emulation is a common practice for achieving interoperability in information systems and can take several different forms (conversion, transformation, translation etc).

In our `README.TXT` example (Section 2), the module t_{README}, which normally requires knowledge of the English language (t_{ENG}), can alternatively be read using t_{GR} (knowledge of the Greek language), provided that we use some translator tool, or a dictionary t_{ENG2GR}, that would translate it in Greek. In this example, t_{ENG} is the emulated module, t_{ENG2GR} is the emulator and t_{GR} is the emulation target.

As described in Section 2, the net effect of this emulation process is that dependencies involving t_{README} are captured by the pair: $\{t_{README}\} > \{t_{ENG}, t_{GR}\}$ and $\{t_{README}\} > \{t_{ENG}, t_{ENG2GR}\}$. Notice that the emulation process causes t_{README} to be no longer dependent on the emulated module (t_{ENG}) alone; also t_{README} does not depend on the emulation scheme (*i.e.*, the emulator t_{ENG2GR} or the emulation target t_{GR}) alone.

In the general case, emulation can take more complex forms: a module t may depend on a number of nodes, some of which may be replaceable by an emulation scheme, which may also consist of a number of nodes. Thus, in its most general form, the "emulated module" and the "emulation scheme" can be families of nodes, say $\mathbf{S}_1, \mathbf{S}_2$ respectively, but the general idea is the same.

Formally, $\mathbf{S}_2 = \{S_{21}, \ldots, S_{2m}\}$ will be called an *emulation scheme* for $\mathbf{S}_1 = \{S_{11}, \ldots, S_{1n}\}$ with respect to t iff:

- For all $i = 1, \ldots, n$, it is *not* the case that $\{t\} > S_{1i}$.
- For all $i = 1, \ldots, m$, it is *not* the case that $\{t\} > S_{2i}$.

– For all $i = 1, \ldots, n$, $j = 1, \ldots, m$, it holds that $\{t\} > S_{1i} \cup S_{2j}$.

In the README.TXT example, we have: $t = t_{README}$, $\mathbf{S}_1 = \{\{t_{ENG}\}\}$, $\mathbf{S}_2 = \{\{t_{ENG2GR}\}, \{t_{GR}\}\}$. Emulation schemes are quite useful in preservation because every user or system that has access to either $\mathbf{S}_1$ or $\mathbf{S}_2$ will be able to understand the content of t; thus, whenever some member of the family $\mathbf{S}_1$ is close to becoming obsolete, it makes sense to consider using (or developing) an emulation scheme for it (with respect to all the interesting modules in our system), so as to retain their intelligibility when $\mathbf{S}_1$ becomes obsolete. Notice that this idea can also be used to model migration, where the role of the emulator in that context is played by the software (module) that applies the migration.

5 Handling Changes

Modules and dependencies may change over time and such changes should be supported. For this reason we describe a number of operations to handle change; such operations can be exploited for defining a protocol between a preservation information system and its users (information providers and consumers).

Following the general trend in fields dealing with changes, we define two general classes of operations: *atomic* and *complex* [6]. Atomic operations are simple, fine-grained operations, whereas complex are more coarse-grained operations, being decomposable into a set of atomic ones. Complex operations usually represent some intuitive and frequently performed type of change, while atomic ones represent some trivial change. Atomic operations are used as “building blocks”, in terms of which more complex operators are built, thus facilitating the definition of the semantics of a complex change. Moreover, atomic operations allow the engineer to override the default behavior of some complex change whenever necessary. Notice that, in principle, any sequence of atomic operations can be considered a complex one, so there is no limit on the number of complex operations that can be defined.

A change operation may cause all sorts of problems upon the related structures. To avoid this, two conditions must be verified following a change:

1. All related structures should be *valid*. Valid means that all dependencies ($>$) in Γ are editable and refer to nodes from $2^{\mathcal{T}}$; moreover, all profiles should contain known modules only, *i.e.*, modules in $\mathcal{T}$ (so: $T_s \subseteq \mathcal{T}$, $T_u \subseteq \mathcal{T}$).
2. The system’s and users’ profiles should be self-intelligible.

There are several options regarding the correct reaction if one of the above conditions fail; our policy is the following: if the first condition fails (invalidity), the change should be either blocked, notifying the engineer of the issue, or some side-effects should be spawned to render the validity condition true (the exact reaction depends on the type of invalidity); if the second condition fails (self-intelligibility), a notification should be issued to the engineer and/or the respective profile owner in order to correct the situation. Non-self-intelligibility is not handled automatically because (a) it is not a very severe problem and (b) there is no single way to restore it, so whatever automatic method we may devise is potentially problematic for certain applications.

5.1 Atomic Changes

Atomic changes should handle changes in modules and dependencies. For modules, we should consider addition and deletion in each of $\mathcal{T}, T_s, T_u$; only the addition and deletion of atomic modules is considered, as it makes no sense to add (or delete) a generalized module (*e.g.*, $\{t_1, t_2\}$). Regarding dependencies, for reasons explained in Section 2, only editable dependencies will be amenable to change, so we consider the addition and deletion of editable dependencies from the graph Γ. This gives a total of 8 atomic operations to consider. All other operations, including replacement, will be handled by complex operations (see subsection 5.2). Table 1 shows the change operations (atomic and complex) considered in this paper and where each one is applied.

Table 1. Change Operations

Operation	Applicable on		
	$\mathcal{T}$	T_s	T_u
$Add_Mod(t)$	•	•	•
$Del_Mod(t)$	•	•	•
$Add_Dep(t, S)$	•		
$Del_Dep(t, S)$	•		
$Replace_Mod(t_1, t_2)$		•	•
$UpgradeBackComp_Mod(t_1, t_2)$	•		
$AddEmulScheme(t, \mathbf{S}_1, \mathbf{S}_2)$	•		

Adding modules to the model ($\mathbf{Add_Mod_T(t)}$). This operation is applicable when a new module is created, or when we learn the existence of a module that was previously unknown; in such cases, a new module (t) is added in $\mathcal{T}$. This operation cannot cause invalidity or self-intelligibility problems (so it has no side-effects). It is rarely executed alone; usually, the new module will be associated with a number of dependency arcs with other (generalized) modules, but this kind of information should be added separately, using other operations.
Adding modules to a profile ($\mathbf{Add_Mod_s(t)}$ and $\mathbf{Add_Mod_u(t)}$). These operations are used in order to add a new module (t) in T_s or T_u. They may cause invalidity if t is not already part of the model ($t \notin \mathcal{T}$). Should this be the case, the operation is blocked and the engineer is notified in order to take proper action. The alternative option to deal with this problem would be to automatically add t to $\mathcal{T}$; this may in fact seem more attractive. However, we chose otherwise because the addition of a new module in $\mathcal{T}$ should be authorized by the engineer and anyway accompanied with a number of dependency additions describing the dependencies associated with t (otherwise we end up with an incomplete model). By notifying the engineer on the issue, we invite him to authorize the addition by introducing the module and the relevant dependencies himself before allowing the addition of the new module to the profile. These operations may also cause non-self-intelligibility, in which case the operation should be executed normally, but a notification should be issued to the engineer and/or respective profile owner to correct the situation somehow.

Deleting modules from the model ($\mathbf{Del_Mod_T(t)}$). This operation is applicable when we spot a modeling error, *i.e.*, when a non-existing module is modeled in Γ; it is not applicable to obsolete modules, as such modules should not be removed from Γ, but from the respective profiles.

This operation may cause invalidity, because there may be dependency arcs involving the deleted module, or involving a generalized module that includes it; moreover, the module may belong to some profile. Thus, the following actions (side-effects) should be taken along with the module deletion, in this order:

1. For each editable dependency of the form $\{t'\} > S$, where $t \in S$, $S \neq \{t\}$ and $t' \neq t$, add a new dependency $\{t'\} > S - \{t\}$ (see operation *Add_Dep* below). Formally, the executed operation is $Add_Dep(t', S - \{t\})$.
2. Delete all editable dependencies of the form $\{t'\} > S$ where $t \in S \cup \{t'\}$ (see operation *Del_Dep* below). Formally, the executed operation is $Del_Dep(t', S)$.
3. Delete module t from the system's and users' profiles (see operations Del_Mod_s and Del_Mod_u respectively below). Formally, the operations executed are $Del_Mod_s(t)$ and $Del_Mod_u(t)$ for all users u, respectively.
4. Upon execution of side-effects (steps 1-3), module t can be deleted from $\mathcal{T}$.

Notice that the removal of t from the profiles (step 3) should be accompanied with a notification to the respective profile owner that module t was non-existent and is removed from the model. Normally, this should not be an issue, as no user could have claimed to have access to a non-existent module, unless he did so by mistake. Also, note that the fact that other operations (side-effects) are executed along with $Del_Mod_T(t)$ does not classify $Del_Mod_T(t)$ as a complex operation, as there is no other atomic operation that can handle step 4 above. An operation having side-effects is different from an operation being decomposable into an equivalent sequence of other operations.

Deleting modules from a profile ($\mathbf{Del_Mod_s(t)}$ and $\mathbf{Del_Mod_u(t)}$). These operations are used in order to delete a non-existent or obsolete module (t) from a profile (T_s or T_u); they may cause non-self-intelligibility, in which case a notification should be issued after the execution of the operation, as usual.

Adding dependencies to the model ($\mathbf{Add_Dep(t, S)}$). This operation is used to add a new editable dependency ($\{t\} > S$) to the model. This operation is useful when a new dependency is created (*e.g.*, as part of the addition of a new module), or when we learn about a previously unknown dependency.

Before executing this operation, it should be verified that the dependency to be added ($\{t\} > S$) is editable and that $\{t\} \cup S \subseteq \mathcal{T}$, *i.e.*, only already known modules are used. Should this be the case, the addition of the dependency can proceed normally, and no invalidities can occur; in a different case, we should reject the operation, as it would cause an invalidity. Moreover, the addition of the new dependency could render the system's and/or some users' profiles non-self-intelligible; as usual, this problem is handled by issuing a notification.

Deleting dependencies from the model ($\mathbf{Del_Dep(t, S)}$). This operation is applicable when we realize that an existing editable dependency is not really true; it is also useful (as a side-effect) when a module is deleted from $\mathcal{T}$. As

usual, only editable dependencies can be deleted. This operation cannot cause invalidity or self-intelligibility problems (so it has no side-effects).

5.2 Complex Changes

In this subsection we define a number of complex operations that we consider useful; as already mentioned, such a list cannot possibly be complete. All operations will be defined in terms of the atomic operations of the previous subsection; notice that the order of execution may be important. The various atomic operations should be performed in a transactional manner, *i.e.*, if one operation in the list fails, the whole complex operation fails and should be rolled back.
Replacing a module in a profile ($\mathbf{Replace_Mod_s(t_1, t_2)}$ and $\mathbf{Replace_Mod_u(t_1, t_2)}$). These two operations are used in order to replace a module t_1 with t_2 in T_s or T_u and are especially useful when a particular module is becoming obsolete and is being replaced (*e.g.*, by a newer version). A replacement consists of a deletion of t_1, followed by the addition of t_2 in the profile; the model $\mathcal{T}$ is not affected. Any more sophisticated functionality should be captured using other operations. If t_2 is in the respective profile, or if t_1 is not, then the operation is rejected; otherwise the following actions should be taken to implement these operations:

1. Perform the operation $Add_Mod_s(t_2)$ (or $Add_Mod_u(t_2)$).
2. Perform the operation $Del_Mod_s(t_1)$ (or $Del_Mod_s(t_1)$).

Upgrading a module with a backwards compatible version in the model ($\mathbf{UpgradeBackComp_Mod(t_1, t_2)}$). Often, modules (*e.g.*, software applications) are being upgraded; such upgrades (new versions) are handled as new modules in our model. However, in many cases, the newer and the older version of the module share some properties, such as dependency relations. To save the engineer from the burden of defining such dependency relations whenever a new, backwards compatible version of a module is inserted, we offer the operation $UpgradeBackComp_Mod(t_1, t_2)$.

This operation adds a new module (t_2) in $\mathcal{T}$ and automatically creates a number of dependencies involving t_2, based on the information on the dependencies involving t_1. In particular, any module depending on t_1 should now depend on t_1 *or* t_2; in addition, t_2 should depend on all (generalized) modules that t_1 depends on. This behavior is justified by the fact that a backwards compatible version normally depends on the same generalized modules as the older version, and can be used as an alternative (to t_1) way of understanding a module; any possible deviation from this default behavior should be captured using additional operations that would restore the desired behavior, overriding the default one.

This operation presupposes that t_1 is already in $\mathcal{T}$ ($t_1 \in \mathcal{T}$), while t_2 is not ($t_2 \notin \mathcal{T}$); if either condition is false, the operation is rejected. Following this operation, the old and the new version (t_1, t_2) will coexist in the graph. The following actions should be taken to implement it:

1. Perform the operation $Add_Mod_T(t_2)$.

2. For each editable dependency $\{t_1\} > S$ do: $Add_Dep(t_2, S)$.
3. For each editable dependency $\{t\} > S$, for which $t_1 \in S$, do: $Add_Dep(t, S \cup \{t_2\})$.
4. For each editable dependency $\{t\} > S$, for which $t_1 \in S$ and $t_2 \notin S$, do: $Del_Dep(t, S)$.

Following the successful execution of this operation, the users (and the system) should be notified on the existence of a new, backwards compatible version of t_1; this might motivate many users (or the system) to upgrade.
Adding an emulation scheme to the model ($\mathbf{AddEmulScheme(t, S_1, S_2)}$). This operation is used to denote that $\mathbf{S}_2$ is an emulation scheme for $\mathbf{S}_1$ with respect to t. The structures $\mathbf{S}_1, \mathbf{S}_2$ are, as usual, families of nodes. This operation automatically determines the relevant dependency changes and executes them, saving us from the burden of updating all the dependencies manually.

Let $\mathbf{S}_1 = \{S_{11}, \ldots, S_{1n}\}, \mathbf{S}_2 = \{S_{21}, \ldots, S_{2m}\}$; it is assumed, as usual, that all related modules (*i.e.*, t and those in $\mathbf{S}_1, \mathbf{S}_2$) are already in the graph (*i.e.*, that $\{t\} \cup S_{11} \cup \ldots \cup S_{1n} \cup S_{21} \cup \ldots \cup S_{2m} \subseteq \mathcal{T}$) and that t depends on the modules of $\mathbf{S}_1$, *i.e.*, $\{t\} > S_{1i}$ is in the graph for all $i = 1, \ldots, n$. If any of these conditions is not true, the operation is rejected. After the execution of the operation, $\mathbf{S}_2$ should be an emulation scheme for $\mathbf{S}_1$ with respect to t. The following actions should be taken to implement this operation:

1. For all $i = 1, \ldots, n$, $j = 1, \ldots, m$ do: $Add_Dep(t, S_{1i} \cup S_{2j})$.
2. For all $i = 1, \ldots, n$ do: $Del_Dep(t, S_{1i})$.
3. For all $i = 1, \ldots, m$ do: $Del_Dep(t, S_{2i})$.

Other operations. Apart from the above complex operations, one could consider a number of other operations, such as operations on renaming modules, replacing dependency relations, or cleaning up the system (referring to the removal of modules that are no longer necessary for the intelligibility of any module in the system's profile). Such operations can be defined in a similar way and are omitted due to lack of space.

6 Concluding Remarks

Recently, there has been a number of theoretical attempts (like [1], [3]), standards (like OAIS[3]) and ongoing international projects (like CASPAR[4] and PLANETS[5]) dealing with digital preservation, indicating a growing interest on the problem and resulting to the study of several of its aspects, such as the definition of metadata and services for preservation, cost-related strategies for data preservation planning etc.

In this paper, we formalized the notions of profile, intelligibility, emulation and evolution based on the notion of dependency and described the services

[3] OAIS reference model (ISO:14721:2003)
[4] http://www.casparpreserves.eu/
[5] http://www.planets-project.eu/

that should be supported by a modern information preservation system. Dependency management has been a subject of research in several (old and newly emerged) areas, from software engineering [4], [9], [10], [11] to ontology engineering [5], [7]; to the best of our knowledge, this is the first paper that uses these notions for digital preservation, so it is quite different from other theoretical attempts on the problem ([1], [3]). Issues for further research include measuring computational complexity and extending the model with complex dependencies, composite modules and dependencies of different granularity.

Acknowledgements

This work was partially supported by the EU project CASPAR (FP6-2005-IST-033572) which aims at building a pioneering framework to support the end-to-end preservation lifecycle for scientific, artistic and cultural information.

References

1. J. Cheney, C. Lagoze, and P. Botticelli. "Towards a Theory of Information Preservation". In *Procs of the 5th European Conf. on Research and Advanced Technology for Digital Libraries, ECDL '01:*, pages 340–351, London, UK, 2001.
2. T. Eiter and G. Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42(1):3–42, January 1995.
3. G. Flouris and C. Meghini. Steps towards a theory of information preservation. In *Proceedings of the International Workshop on Database Preservation*, 2007.
4. X. Franch and N.A.M. Maiden. Modeling Component Dependencies to Inform their Selection. In *2nd Intern. Conf. on COTS-Based Software Systems*, 2003.
5. M. Jarrar and R. Meersman. Formal Ontology Engineering in the DOGMA Approach. *Intern. Conf. on Ontologies, Databases and Applications of Semantics (ODBase)*, pages 1238–1254, 2002.
6. H. Stuckenshmidt and M. Klein. Integrity and change in modular ontologies. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 2003.
7. E. Sunagawa, K. Kozaki, Y. Kitamura, and R. Mizoguchi. An Environment for Distributed Ontology Development Based on Dependency Management. *Proc. of the 2nd Intern. Semantic Web Conference (ISWC2003)*, pages 453–468, 2003.
8. Y. Tzitzikas. "Dependency Management for the Preservation of Digital Information". In *Procs of the 18th Intern. Conf. on Database and Expert Systems Applications, DEXA'2007*, Regensburg, Germany, September 2007. Springer-Verlag.
9. M. Vieira, M. Dias, and D.J. Richardson. Describing Dependencies in Component Access Points. *Procs of the 23rd Intern. Conf. on Software Engineering, ICSE'01, Toronto, Canada*, pages 115–118, 2001.
10. M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. *ASE*, 00:241, 2002.
11. M. Walter, C. Trinitis, and W. Karl. OpenSESAME: an intuitive dependability modeling environmentsupporting inter-component dependencies. *Procs of 2001 Pacific Rim International Symposium on Dependable Computing*, pages 76–83, 2001.