Semantic Version Management based on Formal Certification

Jean-Yves Vion-Dury and Nikolaos Lagos

Xerox Research Centre Europe, 6 chemin de Maupertuis, 38330 Meylan, France

Keywords: Semantic Versioning, Version Management, Software Versioning.

Abstract: This paper describes a Semantic Version Management method that enables managing consistently digital resources throughout their life cycle. The core notion is that resources are described be means of logical specifications formally expressed using an extensible logical language. A new version is considered *certified* only if the resource owner is able to formally prove that it satisfies its logical specification. The method includes formal proofs for qualifying changes (occurring either on the resource content or on the corresponding specifications) and accordingly characterizing them via the definition of appropriate version labels. Based on the above method, a service-oriented solution is also described that enables managing changes consistently, in a sound manner, for both resource owners and users.

1 INTRODUCTION

Versioning is particularly challenging in dynamic, non-centralized architectures, as propagating changes in a consistent manner requires strict coordination of tiers. Such a case is for example service-oriented architectures, where the separation of the service interface from its implementation, often referred to as opacity, is an additional complexity (changes in the interface or the implementation may impact the contract that the service provider has with a consumer from a functional or quality-of-service (QoS) point) (Novakouski12).

This work presents a method towards Version Management that enables consistent management of digital resources throughout their life cycle for the benefit of both resource owners/providers (the entities offering the resource) and resource users (the entities using the resources), and exemplifies it in a service-oriented setting. As part of the method we define how resources can be associated with logical specifications written in an extensible logical formalism understood and agreed by tiers. We also describe how a service can verify the well-formedness of proofs and properties when required, so that the version labels stay consistent[1].

---

[1]For the sake of space only a small number of evolutions are described in this paper. However, all possible evolutions are defined and available online (Vion-Dury15) for interested readers.

2 RELATED WORK

Version management includes defining strategies for: version naming (which should aid in recording the evolution of a resource); and for identifying/responding to different updates (which should be related to controlling the consistency of different versions). In this work we deal with both of the above issues and integrate them into a single framework.

Version naming is most typically designed to reflect consumer compatible and non-compatible (or breaking and non-breaking) changes. The former constitute major releases and the latter minor ones. The corresponding naming usually follows the "Major#.Minor#" scheme where the sequential major release version number precedes the minor one (Jerijrvi08). Alternatively naming schemes may incorporate a time-stamp instead of the above identifier. However, the above naming schemes provide very limited semantic information about the relationships between versions (Conradi98).

Recent work, especially in the area of web services, recognises the importance of having semantic information relevant to versioning and try to inject such information. For instance, (Juric09) extends WSDL and UDDI with corresponding metadata. In (Wetherly13) and (Cacenco06) a three-label version scheme is proposed, where each label is incremented in response to changes in the versioned resource (in their case each label corresponds to one of data components, message components and features).

(Vaivaran09) goes one step further by not only assigning semantics to version labels but also using them to determine the compatibility between entities (e.g. UML processes) and source code modules. However, the naming scheme is defined by the provider and it is the job of the consumers to use this consistently. Our method allows separate identifiers for the service owner/provider and each of the users(s)/consumer(s). Based on this idea we propose the use of a dedicated versioning service implementing our method, which would ensure the consistency of the mapping between the different identifiers.

Similar work can be found, in terms of formalising the evolution of artifacts in dynamic environments, as proposed by (Papazoglou12), where a theoretical type-safe framework is proposed to ensure correct versioning transitions in service-oriented environments. Although their method sketches the relation between the notion of compatibility and specifications, they do not provide any formal grounding on how the modelling of consistent change is reflected by the naming scheme of different variants. In addition, they do not refer at all to the notion of a dedicated versioning service functioning as broker, a direct versioning approach being implied to the best of our understanding. In direct approaches the consumer and provider are directly connected and each one has to manage changes to their requirements and offer respectively. In our case, we propose an intermediary-based approach, where the consumer does not have to communicate with the provider directly but only through a third service (i.e. a broker). The broker is responsible for managing the changes happening to any of the connected services (a preferred approach in the context of service-oriented environments is (Leitner08)).

Another close work is done by (Brada03). The author formalises the notion of component versioning to support consistent substitutability of components and falls under the umbrella of software configuration management for component-based systems. Although very interesting, the work is mainly focused on a specific architecture (widely used for software components) called CORBA (CORBA).

3 THE VERSIONING MODEL

3.1 Invariants, Specifications, Theories

We represent the logical characterisation of a resource via two logical formulae: the first one is the *invariant*, i.e. the logical properties that the resource must always satisfy during its life cycle; the second one,

specification, is a logical formula that may change over time, but that the resource must satisfy in order to be *certified*. The *certification* of a resource is therefore a formally established proof that the new version is consistent with the resource logical specification. Consequently, a version increment can be generated (the choice of the increment type is left to the owner, but must satisfy specific logical properties, as explained later in the paper).

Associating an invariant with a versioned resource captures the fundamental fact that if a resource owner intends to derive versions, then something commons to all versions should be preserved. Otherwise, versions are not needed, one just need to derive a new resource with a different name and different logical characteristics, and start a novel life cycle.

What is the role of the specification ? We assume that versioning systems should have an abstract description that explains what is the resource and how it can be used. For instance, this can be a document describing an API (Software Enginering and Service Oriented Architectures), it can be an SQL schema for a data set, a formal context free grammar for a source code written in a specific programming language, an XSD schema for an XML document, and so on. A specification can also be used to describe tests that a resource should pass in order to be considered valid.

We also distinguish between:

- external specification, describing the behaviour that is expected by users of the resource, e.g. an API exposing methods to a service's clients.

- internal specification, describing the behaviour that is expected by the owner of the resource, e.g. support for changes in the code and documentation.

Such a specification is formally expressed in this work using an underlying core logic $\mathcal{L}$ (see section 4.2.1 and a specific, owner designed, *theory* $\mathcal{T}$, which extends the core logic with axioms and theorems suitable to handle the owner's applicative domain.

The key idea is that the versioning method, presented in this paper, is based on formal proofs to certify a versioning step. Such a proof will take the form of a proof term, i.e a particular data structure that reflects exactly how the proof was constructed from the axioms and the theorem. Although it can be hard to build such a proof, it is easy to verify that it is well-formed and that it is indeed a proof of the claimed property.

3.2 Structure of Version Labels

In our model, a resource r represents a digital content (its digital extension as a bitstream) and is related

to a unique identifier and a version label defined as $[M.m.\mu : \nu]/s$, where M, m, μ, ν, s are natural numbers greater or equal to 0. The number M stands for *major* revision, m for *minor*, μ for *micro*, ν for *variant* and s for *stamp*. Minor versions preserve the backward compatibility (while possibly offering new functionalities); micro versions have no visible impacts from the external specification point of view (improvements, bug fixes, simplifications...); while major versions may require revising the processes depending on the changing resource (this scheme is coined as *Semantic Versioning* (OSGi)). The advantage of this scheme is that it is a straightforward way to communicate to the users of a resource whether its evolutions are backward compatible or incompatible.

The stamp is incremented at each modification operation, independently of versioning mechanisms. As a consequence, a resource can be updated and never versioned, meaning that the changes can be tracked and memorized, but without any assumptions regarding its semantic properties. A resource r is *certified* when a versioning operation was able to logically establish its compliance with its semantic specification(s). The version label is designed in order to reflect this, as the variant component must be null.

3.3 Version based Ordering

Version labels are built in such a way that they can be totally ordered for a given resource, i.e. it is always possible to assess if one version is anterior to another one. This is important since, thanks to this property, one can always find the antecedent of a given version based on the label structure. We define first an order $\prec$ on version labels through:

$$\forall M, M', m, m', \mu, \mu', \nu, \nu', s, s' \text{non negative integers}$$

$$
\begin{aligned}
M < M' &\Rightarrow [M.m.\mu : \nu]/s \prec [M'.m'.\mu' : \nu']/s' \\
m < m' &\Rightarrow [M.m.\mu : \nu]/s \prec [M.m'.\mu' : \nu']/s' \\
\mu < \mu' &\Rightarrow [M.m.\mu : \nu]/s \prec [M.m.\mu' : \nu']/s' \\
\nu < \nu' &\Rightarrow [M.m.\mu : \nu]/s \prec [M.m.\mu : \nu']/s' \\
s < s' &\Rightarrow [M.m.\mu : \nu]/s \prec [M.m.\mu : \nu]/s'
\end{aligned}
$$

Version labels evolve in such a way that the following property always holds.

Proposition 1. *Monotonicity of stamps*

$$\forall M', m', \mu', \nu', s' \text{non negative integers}$$
$$[M.m.\mu : \nu]/s \prec [M'.m'.\mu' : \nu']/s' \Rightarrow s < s'$$

Intuitively, this property says that for a given resource r, the stamp $\mathbf{stamp}(r) = s$ tracks all change history regardless of the evolution of the version label (which is ruled by the preservation of significant semantic properties, as detailed later). In order to abstractly handle resources, we define three so called *destructors* to access those attributes:

$$
\begin{aligned}
\mathbf{identifier}(r) &= \ell \\
\mathbf{content}(r) &= c \\
\mathbf{version}(r) &= [M.m.\mu : \nu]/s
\end{aligned}
$$

Similarly, versioning labels can be de-constructed according to the following functions:

$$
\begin{aligned}
\mathbf{major}([M.m.\mu : \nu]/s) &= M \\
\mathbf{minor}([M.m.\mu : \nu]/s) &= m \\
\mathbf{micro}([M.m.\mu : \nu]/s) &= \mu \\
\mathbf{update}([M.m.\mu : \nu]/s) &= \nu \\
\mathbf{stamp}([M.m.\mu : \nu]/s) &= s
\end{aligned}
$$

Given the above, we can build a partial order over versioned resources as follows.

Definition 1. *partial ordering of resources*

$$\forall r, r' \quad r \prec r' \quad iff \quad \mathbf{identifier}(r) = \mathbf{identifier}(r') \wedge \mathbf{version}(r) \prec \mathbf{version}(r')$$

3.4 Designation and Selection of Resources

To be able to select and retrieve resources, we define herein the designation mechanism. Resources can be designated through a term built from the following syntax:

$$
\begin{aligned}
D &::= \ell \mid \ell[V_1] \mid \ell/s \mid \ell/\star \mid \ell/? \\
V_1 &::= \star \mid ? \mid M \mid M.V_2 \\
V_2 &::= \star \mid ? \mid m \mid m.V_3 \\
V_3 &::= \star \mid ? \mid \mu
\end{aligned}
$$

$?$ denotes all available components that match some versioning constraints, whereas $\star$ denotes the highest available component, and therefore, is always a singleton (or the empty set if the condition is not satisfied). The notation $\ell[?]$ designates the set containing all versions of the resource ℓ, $\ell[\star]$ the last qualified version, $\ell[3.\star]$ designates the last minor and micro version derived from the major revision 3 of ℓ. For some value M, m, μ the designation $\ell[M.m.\mu]$ always maps to a singleton (provided this derivation belongs to the history of the resource), as well as ℓ/s, under the same condition. The notation $\ell/?$ designates the set of all variants of ℓ regardless of any versioning information, and $\ell/\star$ the most recent element of this set, if any.

All resources $\mathcal{E}$ can be retrieved through a resolution function that takes as input the designation term, and returns a set of resources accordingly. The resolution function, noted $!$ is defined in Table 1.

4 THE CERTIFICATION MODEL

4.1 The Dynamics of Resources and Specifications

We consider three different parameters related to a resource that can be changed, according to the model of resources described in previous sections. The resource r, its internal specification I_r, and its external

Table 1: Resolution of references.

reference	result	comment	certified
$!(\ell)$	$!(\ell/\star)$	shorthand	
$!(\ell/?)$	$\{r \in \mathcal{E} \mid \mathbf{identifier}(r) = \ell\}$	all variants	yes/no
$!(\ell/\star)$	$\sup\prec[!(\ell/?)]$	last variant	yes/no
$!(\ell/s)$	$\{r \in!(\ell/?) \mid \mathbf{stamp}(r) = s\}$	a particular variant	yes/no
$!(\ell[M])$	$!(\ell[M.\star])$	shorthand	
$!(\ell[\star])$	$\sup\prec[!(\ell[?])]$	last version	yes
$!(\ell[?])$	$\{r \in!(\ell/?) \mid \mathbf{major}(\mathbf{version}(r)) > 0\}$	all variants	yes
$!(\ell[M.m])$	$!(\ell[M.m.\star])$	shorthand	
$!(\ell[M.\star])$	$\sup\prec[!(\ell[M.?])]$	last M version	yes
$!(\ell[M.?])$	$\{r \in!(\ell/?) \mid \mathbf{major}(\mathbf{version}(r)) = M\}$	all M versions	yes
$!(\ell[M.m.\mu])$	$\{r \in!(\ell[M.m.?]) \mid \mathbf{micro}(\mathbf{version}(r)) = \mu\}$	this version	yes
$!(\ell[M.m.\star])$	$\sup\prec[!(\ell[M.m.?])]$	last $M.m$ version	yes
$!(\ell[M.m.?])$	$\{r \in!(\ell[M.?]) \mid \mathbf{minor}(\mathbf{version}(r)) = m\}$	all $M.m$ versions	yes
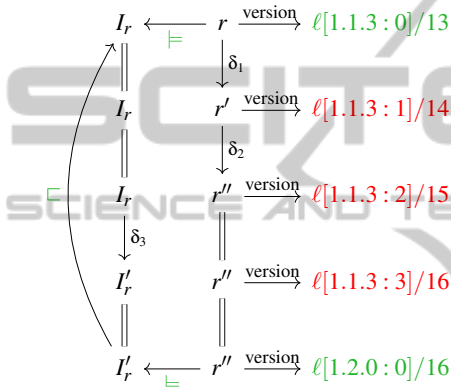
Figure 1: Example updates for a resource r with identifier ℓ and specification I_r.

specification E_r, c. The triple $\langle r, I_r, E_{r,c} \rangle$ may evolve in many different ways, but to be certified, the fundamental property described in section 4.3.1 must always be verified. Still, uncertified resources have a versioning label and therefore can be designated and retrieved. By construction, a version label having a non zero variant is uncertified. For instance, Figure 1 illustrates two evolution steps of a resource r, and then an evolution step of its internal specification I_r (e.g. to take in account added features). These three update operations are applied on a certified resource (with version label $\ell[1.1.3 : 0]/13$). The first two steps track modifications of the resource content, whereas the third one tracks the modification of the internal specification. The last step corresponds to a minor qualification, to establish that the fundamental invariant is satisfied: the resource satisfies the modified specification and this one is a logical extension of the original specification (and is thus backward compatible). From this last property, we can deduce that the modified specification is also a logical extension of the invariant, since the original specification was.

4.2 The Underlying Logic and Its Implementations

The intent of this section is not to define a particular logic suitable to handle the logical mechanisms described in this paper, but, to make clear which qualities are expected from such a logic. The expressiveness of the logic theoretically depends on the requirements of the application; however, the higher order logic is commonly used today, implemented with powerful proof assistants, embedding interactive/automated tactic based theorem provers (Cocquand86; Nipkow02), and is generic enough to cover practical cases.

4.2.1 Logic $\mathcal{L}$

This logic, called $\mathcal{L}$, should be equipped with a particular relation that checks well-formedness of terms, in order to be sure that a logical specification is indeed a term that can be managed by axioms and theorems. We note this relation $\mathbf{wf}(P)$, or equivalently, using a generic typing relation, as $\mathbf{type}(P, prop)$. This logic $\mathcal{L}$ must also be equipped with a proof system able to explicitly handle proof terms. We note $\mathcal{L} \vdash \mathbf{proof}(p, P)$ the logical relation establishing that p is is a proof of P in $\mathcal{L}$. Consequently, proving a property P is a process that will construct a proof term p. In general, it is much more difficult to build the proof term than to check for its correctness, the latter being usually implemented through simple and efficient algorithms (see for instance how proof terms are handled through lambda term in the COC higher order type theory (Cocquand86), how proofs become first order entities in (Kirchner10), and also in OpenTheory (Hurd11) and Dedukti (Boespflug12; Saillard13) which proposes a universal and pivotal proof handling mechanism; more generally, see how proof terms are built and computed in any logic based on the Curry-

Howard correspondence (De Bruijn95)).

We expect also that extensions suited to domain specific applications will be expressed as additional theories $\mathcal{T}_0, \mathcal{T}_1, \cdots$ that will be designed to work on top of $\mathcal{L}$. In those cases, the $\mathcal{L}, \mathcal{T}_0, \mathcal{T}_1, \cdots \vdash$ **proof**(p, P) will be the natural extension of the notation presented above, to express a proof in the aforementioned set of theories.

To illustrate, let us consider the higher order intuitionistic logic $\mathcal{L}$ (whose associated proof system is based on natural deduction (Laboreo05), where elimination and introduction rules are associated with each construct). If we consider a formula like

$$\forall A. \forall B.((A \wedge B) \Rightarrow B)$$

A proof using sequents and natural deduction could be for instance

This proof can be mapped into a structurally equivalent proof term using all applied rule names

$$I_\forall(I_\forall(I_\Rightarrow(E_\wedge(\in_L, I_\vdash(\in_R (\in_L))))))$$

which can be checked for correctness using a dedicated predicate **proof**(p, P) :

4.2.2 Fundamental Logical Relations and Properties

When a resource r satisfies a logical formula F, this is noted as $r \models F$. The $\sqsubset$ relation expresses strict logical implication, i.e

$$F_1 \sqsubset F_2 \text{ iff } F_2 \Rightarrow F_1 \wedge \neg(F_1 \Rightarrow F_2)$$

whereas $\sqsubseteq$ expresses the logical implication (therefore, is reflexive):

$$F_1 \sqsubseteq F_2 \text{ iff } F_2 \Rightarrow F_1$$

The $\sqcap_F$ relation expresses "partial disjunction", i.e, that a logical intersection F exists between two formulas:

$$F_1 \sqcap_F F_2 \text{ iff } F \sqsubset F_1 \wedge F \sqsubset F_2 \wedge \neg(F_1 \sqsubseteq F_2)$$

Defined this way, this relation produces two subcases: one where $F_2 \sqsubset F_1$ (F_2 is strictly more specific than

F_1) and one where $\neg(F_2 \sqsubset F_1)$ (F_1 and F_2 are strictly conjunctive). Both situations make sense when interpreting major version evolution (which potentially can raise incompatibility issues). The first one expresses cases where a specification is restricted (producing some regression in the expectation), the other one expresses situations where a specification evolves by extending some points, but regressing on others.

The $\approx$ relation just expresses logical equivalence (but is typically used when F_1 and F_2 are syntactically distinct):

$$(F_1 \approx F_2) \text{ iff } F_2 \Leftrightarrow F_1$$

It is easy to show that the $\sqsubset, \sqsubseteq, \approx$ relations are transitive, and since the logic is expected to be sound, we also assume the following property holds:

$$(F_1 \sqsubseteq F_2 \wedge r \models F_2) \Rightarrow r \models F_1$$

We denote the change of the resource r into r' by a particular δ relation, such that $\delta(r) = r'$.

## 4.3 Certification of Resources and Specifications

When a client asks that a resource r is certified (creation of a new major, minor or micro version), he/she must provide one or several proofs, depending on the past operations and the current context (some cases are detailed in section 4.3.2). The proofs are checked for correctness, i.e, the system verifies that they indeed establish the expected properties P_i or $\mathcal{L}, \mathcal{T}_0, \mathcal{T}_1, \cdots \vdash$ **proof**(p, P_i). Moreover, depending on the kind of change, some "structural" properties may be verified: well-formedness of changed specifications and that the resource satisfies the specifications.

Checking a proof is easy, while building the proof term (formally proving a property) can be of unbounded difficulty, depending on the application domain (and consequently on the power of theories operated by the resource owners). However, many useful scenarios can be captured by simple theories, and proofs can be established either by a proof assistant or even by specialized programs, such as type checkers, or specialized algorithms. Note that run-time execution performance of such programs would not be as crucial as during development cycles, since they are involved only during the certification phase. Properties such as schema membership (XML validation, database relational schema) can be processed automatically by dedicated algorithms, as long as those schemes are translated into $\mathcal{L}$ inside an appropriate theory. Similarly, API signatures for a software library, making use of decidable type systems, can be similarly processed through an equivalent approach.

Section 4.3.2 details some change cases with respect to the versioning certification. The changes involve three entities: the resource r, its internal specification I_r, and its external specification $E_{r,c}$ for a particular client c.

4.3.1 Version Consistency

The certification process aims at controlling the evolution of the tuple $\langle r, \mathbf{I}_r, I_r, E_{r,c} \rangle$ for all resources r, for all clients c of r and associated specifications I_r and $E_{r,c}$, and the invariant $\mathbf{I}_r$. This is achieved in a way that the resource satisfies all specifications, and the external specification visible by the client is a specialization of the internal specification (controlled by the owner). Both external and internal specifications must always specialize the initial invariant $\mathbf{I}_r$. This fundamental scheme is illustrated in the following diagram.
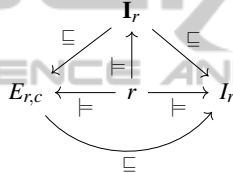
4.3.2 Version Co-evolution

We examine changes potentially affecting any of the three dynamic parameters (therefore, excluding the invariant $\mathbf{I}$), focusing on the consequences both from the clients' and owner's perspectives, in terms of how their respective version labels will evolve. The elements with red color correspond to the logical relations that must be established, whereas the green ones express what can be deduced both from the previously established relations and from the proved statements (and therefore, do not need require formal treatment).

As many combinations exist, we explicit here only two cases: the first one deals with unchanged resources but evolving specifications (no change regarding the resource itself, but changes to the corresponding specifications), while in the second case the resource changes as well. See (Vion-Dury15) for the exhaustive list of cases.

Update describing Changing Specifications and unchanged Resources: The figure 2 shows a certification scheme where the client considers a change as having a major effect.

Update with Changing Resources: We distinguish three cases, from the point of view of a client c, working with an external specification $E_{r,c}$ over a resource

Figure 2: Update with changing specifications and unchanged resource: client sees a major evolution (M, m and μ stands for major, minor, micro).

r. Figure 3 covers major changes with an evolving external specification.

5 VERSION BROKER SERVICE

This section presents a service-oriented application setting for the method described in the previous sections. We consider that a specialised broker can offer services related to versioning (and generally resource lifecycle management) by implementing our method as described herein. We call that service the Version Broker Service (VBS).

We distinguish between two roles for the service's clients: the resource owner and the resource user. The resource owner publishes one or several logical descriptions, at various levels of precision, which characterize the resource and will serve as a basis for the formal agreement about its intended use. Users may subscribe to a "contract" (e.g. Quality of Service agreement) and get access to the resource through a dedicated version management scheme. Each time the owner certifies a new version, the VBS will make sure that this change is propagated to the "contracts", i.e that each version label associated with the contracted interface is derived in a semantically consistent way. In the remaining part of the section we describe in more detail what the service offers but also requires from the owners and users of the resources.
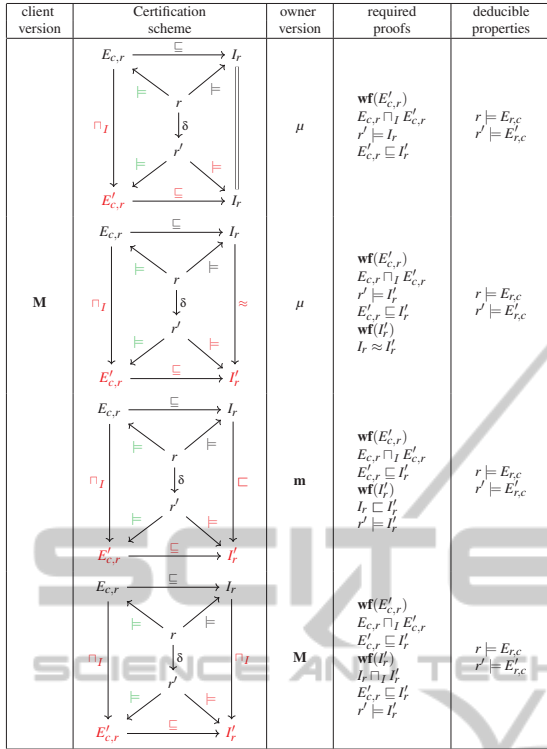
client version	Certification scheme	owner version	required proofs	deducible properties
	(diagram)	μ	$\mathbf{wf}(E'_{c,r})$ $E_{c,r} \sqcap_I E'_{c,r}$ $r' \models I_r$ $E'_{c,r} \sqsubseteq I'_r$	$r \models E_{r,c}$ $r' \models E'_{r,c}$
	(diagram)	μ	$\mathbf{wf}(E'_{c,r})$ $E_{c,r} \sqcap_I E'_{c,r}$ $r' \models I'_r$ $E'_{c,r} \sqsubseteq I'_r$ $\mathbf{wf}(I'_r)$ $I_r \approx I'_r$	$r \models E_{r,c}$ $r' \models E'_{r,c}$
M	(diagram)	m	$\mathbf{wf}(E'_{c,r})$ $E_{c,r} \sqcap_I E'_{c,r}$ $E'_{c,r} \sqsubseteq I'_r$ $\mathbf{wf}(I'_r)$ $I_r \sqsubset I'_r$ $r' \models I'_r$	$r \models E_{r,c}$ $r' \models E'_{r,c}$
	(diagram)	M	$\mathbf{wf}(E'_{c,r})$ $E_{c,r} \sqcap_I E'_{c,r}$ $E'_{c,r} \sqsubseteq I'_r$ $\mathbf{wf}(I'_r)$ $I_r \sqcap_I I'_r$ $E'_{c,r} \sqsubseteq I'_r$ $r' \models I'_r$	$r \models E_{r,c}$ $r' \models E'_{r,c}$

Figure 3: Major changes with an evolving external specification (M, m and μ stands for major, minor, micro; δ denotes a modification of the resource).

5.1 The Resource Owner

The resource owner creates the resource and defines how it evolves. This includes deciding when and how to publish the resource and also what type of access and use should be allowed for different users. The owner also defines the domain specific theories needed (or use/extend those that might be proposed by the VBS), and design the properties needed to characterize the provided resource. The resource extension (its bitstream) might be directly uploaded into the VBS inner storage space, or just localized through a pointer that allows the VBS to access the content when needed. The actions that the VBS allows/expects from the resource owner are:

Theory creation. Upload a source file compatible with the underlying logic (see section 4.2) supported by the VBS, and an associated name that must be unique for this owner. This means that the VBS has a dedicated parser and computational means to verify/type the provided definitions (check for well-formedness). Once loaded and verified by the VBS the definitions can be used in other operations.

Resource creation. Allows to specify a name uniquely associated with the target resource. This name will be used to designate the revisions thanks to the version labels, as in *cobra.socket.api*, which could be subsequently referenced as in e.g. *cobra.socket.api*[1.2.0].

Internal specification creation. VBS expects the resource name to which the specification will be attached, and a logical formula, which will be checked for well-formedness. The VBS returns a unique identifier, so that the specification can be designated without ambiguity for future operations. *External specification creation.* idem (see above).

Resource update. Upload a novel extension for the resource, or equivalently, apply a patch; a new internal version label is computed accordingly (see *update* in Table 2).

Specification update. Upload a novel extension for the resource (or equivalently, apply a patch); a new internal version label is computed accordingly (see *update* in Table 2).

Qualification. This corresponds to the first stage involving a logical characterization. Once certified, the resource can be derived and made visible to external users. As parameters, the VBS expects the resource label $\ell = \mathbf{identifier}(r)$, a reference to the invariant property I_ℓ and to the internal specification S_ℓ, and the following proofs (see 4.3.1): (*i*) the invariant is implied by the internal specification in the defined context $\mathcal{L}, \mathcal{T}_i \vdash I_\ell \sqsubseteq S_\ell$ and (*ii*) the resource satisfies the internal specification in the defined context $\mathcal{L}, \mathcal{T}_i \vdash \ell \models S_\ell$ The VBS then verifies the proofs and if correct, qualifies the resource, generating the first certified version label (see Table 2).

Derivation. Compute a new version for a resource, whose name is given as a parameter, as well as the type of version: micro, minor, or major. Depending on the version type, different proofs will be required by the VBS, so as to minimize the proving cost (see figure 4).

Publication. Requires the resource's name as parameter and synchronizes the last certified version to generate the external version labels for all users. Proofs will be asked by the VBS according to the co-evolution schemes (see section 4.3.2).

In the following, we use x^+ to denote $x+1$. Table 2 defines the transformations of the version label according to the operation done on a resource r, where $version(r) = [M.m.\mu : \nu]/s$:

5.2 The Resource User

The resource user accesses the bitstream extension (the resource content) corresponding to particular versions, thanks to the designation mechanism described in section 3.4. The corresponding external specification may also be accessed.

Table 2: Operations supported by the Version Broker Service and corresponding impact on version labels.

operation on resource	version label (before)	version label (after)	certified
creation	$n.a.$	$[0.0.0:1]/0$	no
qualification	$[0.0.0:\nu]/s$	$[1.0.0:0]/s$	yes
major derivation	$[M.m.\mu:\nu]/s$	$[M^+.0.0:0]/s$	yes
minor derivation	$[M.m.\mu:\nu]/s$	$[M.m^+.0:0]/s$	yes
micro derivation	$[M.m.\mu:\nu]/s$	$[M.m.\mu^+:0]/s$	yes
update	$[M.m.\mu:\nu]/s$	$[M.m.\mu:\nu^+]/s^+$	no

Figure 4: Certified evolution of a resource ℓ according to a specification S (δ denotes a modification of the resource).

The operations that the VBS supports for users of the resource include:

Resource download. VBS expects a full version label (e.g. $dp.api[2.1.3]$), and returns a set of resource contents, associated with its exact version label. Such a label will resolve into this particular content if a subsequent access is required. As an example, a first request with $db.api[1.2.?]$ may return $\{(c_1, db.api[1.2.1]), (c_2, db.api[1.2.2])\}$ (where c_i denotes the respective bitstream contents) whereas a $db.api[1.2]$ request will be interpreted by the VBS as $db.api[1.2.\star]$ and would return the last content available for this major and minor version in the given context, i.e. the singleton $\{\langle c_1, db.api[1.2.2]\rangle\}$.

Specification download. Using the same communication scheme, the resource user may access each external specification associated with each version resolved by the VBS.

6 ILLUSTRATIVE EXAMPLE

In this section we present an example that illustrates the concepts introduced in the previous sections. We assume that a provider manages a software library offering persistent storage operations. We call the library *STO* and assume that it is made available as a compiled module. The owner of *STO* chooses to expose two different views of this library: a basic one including creating/opening a persistent store, storing, deleting, and retrieving data, and a more complex one that can additionally handle transactions. We assume that the owner of *STO* uses an object-oriented dedicated theory to describe the resource, focusing on the naming and typing of classes and methods.

6.1 Fundamental Invariant

The fundamental invariant of *STO* is defined in terms of a class, three methods to deal with data, and three basic functions to create, open and delete a database.

$$
\begin{aligned}
\mathbf{I}_{STO} \equiv \quad & \mathbf{class}(\text{STOHandler}) \\
\wedge \quad & \mathbf{language}(\text{Python2}) \\
\wedge \quad & \mathbf{method}(\text{STOHandler}, \text{store}) \\
\wedge \quad & \mathbf{method}(\text{STOHandler}, \text{retrieve}) \\
\wedge \quad & \mathbf{method}(\text{STOHandler}, \text{close}) \\
\wedge \quad & \mathbf{function}(\text{create}) \\
\wedge \quad & \mathbf{function}(\text{open}) \\
\wedge \quad & \mathbf{function}(\text{delete})
\end{aligned}
$$

Note that according to the vision of the resource

owner, signatures of methods/functions are not considered as fundamental, and therefore, could change along the lifetime of the resource, however, the programming language and version must stay stable.

6.2 The External Specification

It describes a minimal view on the functionality offered by the API, namely, the signatures of methods and functions, including potential exceptions as follows.

$$
\begin{aligned}
E_{STO} \quad \equiv \quad & \mathbf{I}_{STO} \\
\wedge \quad & \mathbf{signature}(STOHandler.store, [\mathbf{string}, \mathbf{string}], \mathbf{void}) \\
\wedge \quad & \mathbf{signature}(STOHandler.retrieve, [\mathbf{string}], [\mathbf{string}]) \\
\wedge \quad & \mathbf{signature}(STOHandler.close, [\,], \mathbf{void}) \\
\\
\wedge \quad & \mathbf{signature}(create, [\mathbf{string}], STOHandler) \\
\wedge \quad & \mathbf{signature}(delete, [\mathbf{string}], \mathbf{void}) \\
\wedge \quad & \mathbf{signature}(open, [\mathbf{string}], STOHandler) \\
\\
\wedge \quad & \mathbf{throw}(create, \mathbf{Exception}) \\
\wedge \quad & \mathbf{throw}(delete, \mathbf{Exception}) \\
\wedge \quad & \mathbf{throw}(open, \mathbf{Exception})
\end{aligned}
$$

6.3 The Internal Specification

This one is typically more complex, as expected to help the designer maintaining his source code in a coherent way while hiding (potentially irrelevant) complexity to users. To illustrate this, we decided to consider three additional methods: *check* (verify the presence of a data in the store), *replace* (change a value using a known key) and *put* (store one or many values using the same key). Those methods may be used internally to build higher level operations such as the *store* method which perform storage with replacement, as in standard dictionary data structures based on a hashing algorithm.

$$
\begin{aligned}
I_{STO} \quad \equiv \quad & \mathbf{I}_{STO} \\
\wedge \quad & \mathbf{signature}(STOHandler.store, [\mathbf{string}, \mathbf{string}], \mathbf{void}) \\
\wedge \quad & \mathbf{signature}(STOHandler.retrieve, [\mathbf{string}], [\mathbf{string}]) \\
\wedge \quad & \mathbf{signature}(STOHandler.close, [\,], \mathbf{void}) \\
\\
\wedge \quad & \mathbf{signature}(create, [\mathbf{string}], STOHandler) \\
\wedge \quad & \mathbf{signature}(delete, [\mathbf{string}], \mathbf{void}) \\
\wedge \quad & \mathbf{signature}(open, [\mathbf{string}], STOHandler) \\
\\
\wedge \quad & \mathbf{method}(STOHandler, check) \\
\wedge \quad & \mathbf{method}(STOHandler, replace) \\
\wedge \quad & \mathbf{method}(STOHandler, put) \\
\wedge \quad & \mathbf{signature}(STOHandler.check, [\mathbf{string}], \mathbf{integer}) \\
\wedge \quad & \mathbf{signature}(STOHandler.replace, [\mathbf{string}, \mathbf{string}], \mathbf{void}) \\
\wedge \quad & \mathbf{signature}(STOHandler.put, [\mathbf{string}, \mathbf{string}], \mathbf{void}) \\
\\
\wedge \quad & \mathbf{class}(\mathbf{PathException}) \\
\wedge \quad & \mathbf{subtype}(\mathbf{PathException}, \mathbf{Exception}) \\
\wedge \quad & \mathbf{throw}(create, \mathbf{PathException}) \\
\wedge \quad & \mathbf{throw}(delete, \mathbf{Exception}) \\
\wedge \quad & \mathbf{throw}(open, \mathbf{PathException})
\end{aligned}
$$

Note also that exception management is more detailed thanks to a class specialization.

6.4 Verification of Specifications

The well-formedness of the logical specification is established by proofs using both a generic (for the basic logic operators) and a dedicated theory. In Figure 5 we just give an idea of what the dedicated theory could be for the illustrative example presented above ($x : y$ is the infix notation of the predicate $\mathbf{type}(x, y)$, and $\mathtt{label}, \mathtt{integer}, \cdots$ are built-in predicates to assess lexical properties of items).

The formal proofs required by the VBS for the three specifications will be $\vdash I_{STO}, \vdash E_{STO}$ and $\vdash I_{STO}$, which can be reduced into $\vdash I_{STO}, \vdash P_1$ and $\vdash P_2$ since $I_{STO} \equiv I_{STO} \wedge P_2$ and $E_{STO} \equiv I_{STO} \wedge P_1$ (P_i being the additional properties illustrated by sections 6.2, 6.3).

6.5 First Certification

After submitting the internal and external specifications, the owner wants to produce a first certification of *STO* (that is, asks the VBS to generate a certified version label $STO[1.0.0 : 0]/0$). To that end, the owner must provide a proof that *STO* satisfies the specification ($STO \models I_{STO}$). The computational characterisation of the proof will depend on the power of the underlying theory, on the properties of the programming language, and on the difficulty of the task (and also on the performance level of the algorithm/human operator).

In the weakest case, the proof is not provided by the owner (in that case the owner assumes responsibility for the consistent use of the resource). Yet, proofs of well-formedness ($\vdash P$), as well as implication and partial disjunction of properties are required to control the quality of specifications and the consistency of the claimed evolution of versions. The absence of strong compliance proofs can be compensated by offering testing infrastructure at the VBS level, and including runtime tests in the specifications, e.g. through dedicated predicates like $\mathbf{test}(context, code, value)$ or $\mathbf{raises}(context, code, exception)$, where *code*, *value* and *exception* are particular expressions of some suitable abstract language.

In the strictest case, the programming language is associated with a formal specification system (e.g based on predicate transformers) able to conduct semi-automatic proofs of correctness and to export proof terms in the VBS compliant form.

For intermediate cases, the programming language can be associated with static analysis tools

$$
\begin{array}{llll}
\forall P. & P\!:\!\textbf{prop} & \Rightarrow & \vdash P \\
\forall x,t. & t\!:\!\textbf{type} & \Rightarrow & (x\!:\!t)\!:\!\textbf{prop} \\
& & & \textbf{void}\!:\!\textbf{type} \\
& & & \textbf{any}\!:\!\textbf{type} \\
& & & \textbf{type}\!:\!\textbf{type} \\
& & & \textbf{prop}\!:\!\textbf{type} \\
& & & \textbf{integer}\!:\!\textbf{type} \\
& & & \textbf{string}\!:\!\textbf{type} \\
\forall t. & t\!:\!\textbf{type} & \Rightarrow & \textbf{list}(t)\!:\!\textbf{type} \\
\forall c. & \textbf{class}(c) & \Rightarrow & c\!:\!\textbf{type} \\
& & & \textbf{class}(\textbf{Exception}) \\
\\
\forall x. & \texttt{integer}(x) & \Rightarrow & x\!:\!\textbf{integer} \\
\forall x. & \texttt{string}(x) & \Rightarrow & x\!:\!\textbf{string} \\
\forall t. & t\!:\!\textbf{type} & \Rightarrow & [\,]\!:\!\textbf{list}(t) \\
\forall e,L,t. & t\!:\!\textbf{type} \Rightarrow e\!:\!t \Rightarrow L\!:\!\textbf{list}(t) & \Rightarrow & [e|L]\!:\!\textbf{list}(t) \\
\\
\forall t_1,t_2. & \textbf{subtype}(t_1,t_2) & \Rightarrow & \textbf{subtype}(\textbf{list}(t_1),\textbf{list}(t_2)) \\
\forall t. & t\!:\!\textbf{type} & \Rightarrow & \textbf{subtype}(t,\textbf{any}) \qquad \prec_{any}\\
\forall x,t_1,t_2. & \textbf{subtype}(t_1,t_2) \Rightarrow x\!:\!t_1 & \Rightarrow & x\!:\!t_2 \\
\forall c_1,c_2,m. & \textbf{subtype}(c_1,c_2) \Rightarrow \textbf{method}(c_2,m) & \Rightarrow & \textbf{method}(c_1,m) \\
\forall t_1,t_2,t_3. & \textbf{subtype}(t_1,t_2) \Rightarrow \textbf{subtype}(t_2,t_3) & \Rightarrow & \textbf{subtype}(t_1,t_3) \\
\forall t. & & & \textbf{subtype}(t,t) \\
\\
\forall x. & \texttt{label}(x) & \Rightarrow & \textbf{class}(x)\!:\!\textbf{prop} \\
\forall x,y. & x\!:\!\textbf{type} \Rightarrow y\!:\!\textbf{type} & \Rightarrow & \textbf{subtype}(x,y)\!:\!\textbf{prop} \\
\forall x. & \texttt{label}(x) & \Rightarrow & \textbf{function}(x)\!:\!\textbf{prop} \\
\forall c,m. & \textbf{class}(c) \Rightarrow \texttt{label}(m) & \Rightarrow & \textbf{method}(c,m)\!:\!\textbf{prop} \\
\forall c,m,i,o. & \textbf{method}(c,m) \Rightarrow i\!:\!\textbf{list}(\textbf{type}) & \Rightarrow & o\!:\!\textbf{type} \\
& & \Rightarrow & \textbf{signature}(c.m,i,o)\!:\!\textbf{prop} \\
\forall f,i,o. & \textbf{function}(f) \Rightarrow i\!:\!\textbf{list}(\textbf{type}) \Rightarrow o\!:\!\textbf{type} & \Rightarrow & \textbf{signature}(f,i,o)\!:\!\textbf{prop} \\
\forall c,m,e. & \textbf{method}(c,m) \Rightarrow e\!:\!\textbf{Exception} & \Rightarrow & \textbf{throw}(c.m,e)\!:\!\textbf{prop} \\
\forall f,e. & \textbf{function}(f) \Rightarrow e\!:\!\textbf{Exception} & \Rightarrow & \textbf{throw}(f,e)\!:\!\textbf{prop} \\
\\
\forall f,c_1,c_2. & \textbf{subtype}(c_1,c_2) \Rightarrow \textbf{throw}(f,c_1) & \Rightarrow & \textbf{throw}(f,c_2) \qquad \prec_{throw}\\
\forall f,c_1,c_2. & (i_1 \Rightarrow i_2) \wedge \textbf{subtype}(o_2,o_1) & \Rightarrow & \textbf{signature}(f,i_1,o_1) \\
& & \Rightarrow & \textbf{signature}(f,i_2,o_2) \qquad \prec_{sig}\\
\forall t_1,t_2,T_1,T_2. & \textbf{subtype}(t_1,t_2) \wedge \textbf{subtype}(T_1,T_2) & \Rightarrow & \textbf{subtype}([t_1|T_1],[t_2|T_2]) \\
& & & \textbf{subtype}([\,],[\,])
\end{array}
$$

Figure 5: Example dedicated theory.

(such as type or property checkers using partial evaluation or model checking).

To perform the first certification, the VBS will require $STO \models I_{STO}$, $I_{STO} \sqsubseteq E_{STO}$ (easy), $I_{STO} \sqsubseteq I_{STO}$ (easy, but longer), and $E_{STO} \sqsubseteq I_{STO}$ (a bit more difficult). The only difficulty in the last one, is about proving properties like e.g. $\textbf{throw}(create, \textbf{PathException}) \Rightarrow \textbf{throw}(create, \textbf{Exception})$, which requires using appropriate subtyping oriented axioms as follows (proof scheme in abbreviated form).

$$
\cfrac{
  \cfrac{
    \cfrac{
      \cfrac{
        \cfrac{\top}{\ } \in_{\vdash L} \\ \vdots
      }{\ } \in_{\vdash R} \\ \vdots
    }{\textbf{subclass}(\textbf{PathException},\textbf{Exception}) \in \gamma} \in_{\vdash R}
  }{\gamma \vdash \textbf{subclass}(\textbf{PathException},\textbf{Exception})} I_{\vdash L}
}{\gamma \vdash \textbf{throw}(create,\textbf{PathException}) \Rightarrow \textbf{throw}(create,\textbf{Exception})} \prec_{throw}
$$

6.6 A Change and Its Co-evolution

To illustrate the notion of co-evolution, we propose to examine a change that occurs in both the resource (modification of source code and, accordingly, of the library resource STO) and in the internal specification. Now, the *store*, *retrieve*, *put* and *replace* methods of the STOHandler class could accept any kind of value, and not only strings. This would constitute a major evolution from the internal side (the new specification is more specific, that is, $\delta(I_{STO}) \sqsubseteq I_{STO}$, whereas the E_{STO} need not to be upgraded. As an example, the proof for the *store* method would be as follows.

$$
\cfrac{
  \cfrac{\ldots}{\textbf{subtype}([string,string],[string,any])} \qquad
  \cfrac{\top}{\textbf{subtype}(void,void)}
}{
\gamma \vdash
\begin{array}{l}
\textbf{signature}(STOHandler.store,[string,string],void) \Rightarrow \\
\textbf{signature}(STOHandler.store,[string,any],void)
\end{array}
}
$$

28

7 CONCLUSION

In this paper we presented a method for versioning that enables managing consistently digital resources throughout their life cycle. The method assigns explicit semantics to version labels and describes resources in terms of properties that can be checked for validity based upon formal logical theories. Only if these properties are valid the resource is marked as *certified*. We also sketched how this method could be implemented in a service-oriented setting. We have shown that following our versioning approach entails benefits to both resource users and owners. The resource users have a strong guarantee with respect to the versioning of certified resources. More specifically, that the versioning scheme always reflects in a consistent way the evolution of the resources they contracted for. On the other hand, the resource owners receive valuable support for coherently managing changes between versions while minimizing the requested proofs at each step.

We plan of extending the above work by introducing, within the same setting, the formal underpinnings for version branching and merging, and to start experimenting with the main concepts presented in this paper. The challenge of scaling such a system to a real environment should be understood more concretely, as well as the level of precision we can expect and manage regarding the various logical specifications involved in a system based on our model.

ACKNOWLEDGEMENTS

REFERENCES

M. Novakouski, G. Lewis, W. Anderson and J. Davenport. *"Best Practices for Artifact Versioning in Service-Oriented Systems,"* Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Note CMU/SEI-2011-TN-009, 2012. resources.sei.cmu.edu/asset_files/TechnicalNote/2012_004_001_15356.pdf

K. Jerijærvi and J.-J. Dubray, *Contract Versioning, Compatibility and Composability,"* InfoQ Magazine, Dec. 2008; www.infoq.com/articlescontract-versioning-comp2

R. Conradi and B. Westfechtel. "Version models for software configuration management". ACM Comput. Surv. 30, 2 (June 1998), 232-282. DOI=10.1145/280277.280280 doi.acm.org/ 10.1145/280277.280280

M. B. Juric, A. Sasa, B. Brumen and I. Rozman, "WSDL and UDDI extensions for version support in web services". Journal of Systems and Software, Volume 82, Issue 8, August 2009, pp 1326-1343. ISSN 0164-1212, dx.doi.org/10.1016/j.jss.2009.03.001. www.science direct.com/science/article/pii/S0164121209000478

Curtis Wetherly, Bryan R. Goring, Michael Shenfield, Michael Cacenco. System and method for implementing data-compatibility-based version scheme, US Patent 8,555,272. 2013.

Cacenco, M. and Goring, B. and Shenfield, M. and Wetherly, C. I*mplementing data-compatibility-based version scheme*, WO Patent App. PCT/CA2005/001,345, 2006.

Vairavan, V. and Bellur, U. Method and system for versioning a software system. US Patent App. 12/324,950, 2009.

M. P. Papazoglou, S. Benbernou, V. Andrikopoulos "On the Evolution of Services," IEEE Transactions on Software Engineering, vol. 38, no. 3, pp. 609-628, May-June, 2012 - preprint, infolab.uvt.nl/ mikep/publications/IEEE-TSE%20%5Bpreprint%5D.pdf

Leitner, P.; Michlmayr, A.; Rosenberg, F.; Dustdar, S. "End-to-End Versioning Support for Web Services," Services Computing, 2008. SCC '08. IEEE International Conference on , vol.1, no., pp 59-66, July 2008 - Technical report version doi: 10.1109/SCC.2008.21 www.infosys.tuwien.ac.at/staff/leitner/papers/TUV-1841-2008-1.pdf

P. Brada. "Specification-Based Component Substitutability and Revision Identification". PhD thesis, *Charles University,* Prague, August 2003, d3s.mff.cuni.cz/publications/download/brada_phd.pdf

CORBA 3.3. Accessed June 26, 2014. www.omg.org/spec/CORBA/3.3/.

Semantic Versioning, Technical Whitepaper, OSGi Alliance, Revision 1.0, May 2010 www.osgi.org

T. Cocquand and G. Huet. "The Calculus of Constructions". INRIA Research Report RR-0530, May 1986.

T. Nipkow, L. C. Paulson and M. Wenzel (ed.). "Isabelle/HOL: a proof assistant for higher-order logic". Springer, 2002.

F. Kirchner and C. Muñoz. "The Proof Monad." *The Journal of Logic and Algebraic Programming* 79.3 (2010): 264-277.

J. Hurd. The OpenTheory Standard Theory Library. In NASA Formal Methods, 17791. Springer, 2011.

M. Boespflug, Q. Carbonneaux and O. Hermant, "The lambda-Pi-calculus Modulo as a Universal Proof Language". In PxTP 2012.

R. Saillard. "Dedukti: a universal proof checker". In: Foundation of Mathematics for Computer-Aided Formalization Workshop. 2013.

N. G. De Bruijn. "On the roles of types in mathematics".

The Curry-Howard isomorphism, 1995, vol. 8, p. 27-54.

D. Laboreo. Introduction to Natural Deduction. Tutorial, May 2005. www.danielclemente.com/ logica/dn.en.pdf.

PERICLES a FP7 European project. 2013-2017. www.pericles-project.eu.

J.-Y. Vion-Dury and N. Lagos, Technical Annex www.xrce.xerox.com/content/download/34443/3724 76/file/SV-ANNEX.pdf.