Semantically Extensible Schemas for Web Service Evolution

Erik Wilde

Swiss Federal Institute of Technology Zürich (ETHZ)

Paper URL: http://dret.net/netdret/publications#wil04j

Abstract. Web Services are designed for loosely coupled systems, which means that in many cases it is not possible to synchronously upgrade all peers of a Web Service scenario. Instead, Web Service peers should be able to coexist in different versions. Additionally, older software versions often could benefit from upgrades to the service if they were able to understand it. This paper presents a framework for semantically extensible schemas for Web Service evolution. The core idea of is to use declarative semantics to describe extensions to a service's vocabulary. These declarative semantics can be used by older software versions to understand the semantics of extensions, thus enabling older software to dynamically adapt to newer versions of the service. As long as declarative semantics are sufficient, older software can benefit from the service's extension.

1 Introduction

In this paper, a framework for semantically extensible schemas for Web Service evolution is presented. It is based on various technologies related to the *Extensible Markup Language (XML)* [1]. The basic idea is to construct a framework which augments the almost non-existent support for versioning of Web Services in the *Simple Object Access Protocol (SOAP)* and the *Web Service Definition Language (WSDL)* [2]. Versioning not only covers controlled ways to deal with different version of a service's vocabulary, but also means to semantically describe extensions, so that older software versions can "understand" newer versions of the service vocabulary.

In general, Web Services claim to be middleware for distributed applications, and they also claim to be particularly well-suited for Web-style scenarios, with the most important characteristic being *loose coupling*. Loose coupling means that the components of such a system should be as independent as possible, and this includes the independent evolution of individual components. In this paper, we present an approach which makes it easier for Web Service designers and users to achieve loose coupling. Whereas we describe this approach as a separate concept, based on certain schema design guidelines, we hope that future versions of Web Service specifications will provide better built-in support for this type of service evolution.

The Web's ongoing development is an excellent example for loose coupling, because for the successful evolution of the Web it is crucial that the components (content providers on the server side and Web browsers as clients) can evolve individually. It is unrealistic to expect that servers and clients are always updated simultaneously, and one of the most important success factors of the Web is its ability to deal with version mismatches in a graceful way. This is described in more detail in Section 2, describing compatibility issues, and Section 4.1, giving an HTML-oriented example of how to implement different strategies for dealing with version mismatches.

Even though Web Services in their most general sense can be regarded as services as ubiquitous and easily accessible as Web servers, in reality most Web Service scenarios are closed application areas, regulated through companies or consortia which make change management much easier than in an open environment. However, as Web Services grow more popular, are used by larger communities, and exist for longer times, service evolution and associated change management will become an important issue.

The general question of how XML vocabularies can be versioned has been discussed in a W3C document by Orchard and Walsh [3] and is still an active field of research. In this paper, the question of vocabulary and schema design is discussed in Section 3, which looks at XML vocabularies from the perspective of XML Schema only. More generally, approaches to improve semantic interoperability through various mechanisms have been published by Su et al. [4], Hunter [5], Castano [6], and others, because semantic interoperability is a interdisciplinary field with many possible applications areas. However, the specific issue of how to leverage semantic interoperability for Web Service evolution so far has not been investigated in detail.

2 Compatibilities

The focus of this paper is how to incorporate features for Web Service evolution into schema design. In order to discuss this issue more specifically, Web Service evolution as the parallel development, deployment, and operation of Web Service servers and clients must be regarded for the two possible types of versioning issues:

- *Backward Compatibility:* This is the easier variant of compatibility, which requires that a newer version of a program can interpret and use the data formats of older versions. Backward compatibility can be implemented fairly easily by requiring that every new revision of a program must be able to deal with the data format implemented by older revisions.
- *Forward Compatibility:* This type of compatibility is harder to achieve, and in many real-life examples is not implemented in a usable way. Forward compatibility means that an older version of a program should be able to interpret and use newer data formats, either by ignoring unknown extensions (*weak forward compatibility*), or by being able to interpret extensions (*strong forward compatibility*).
  The challenge for forward compatibility is to deal with possible future versions without knowing ahead what they will look like. Naturally, this is impossible in an unrestricted scenario, where future versions are unrestricted in their ability to change the data format. However, if the data format is based on a schema that provides reasonable extension points and strategies, then it is possible to achieve forward compatibility.

Translated into the world of Web Services, backward and forward compatibility often appear in parallel, because most Web Services are request/response-based. So if a server is upgraded, then the server must be backward compatible to understand requests from older clients, and clients must be forward compatible to understand the server's responses.

Since backward compatibility is easy to achieve, it is not considered in this paper. Forward compatibility, on the other hand, is a challenging issue and the focus of this paper. In the vast majority of Web Service scenarios, there are many more clients than there are servers. Since the absence of forward compatibility means that a server can only be updated after all clients have been updated, too, it is desirable to avoid this kind of dependency. Compatibility in both directions guarantees a maximum degree of independence of servers and clients, and since the concept of Web Services is based on the idea of loosely coupled systems, this independence is very important.

Forward compatibility means that older clients must be able to interpret data from newer servers, without the requirement of an complex version negotiation and data transformation process. Also, it may be required for clients to actually use new data that has not been part of the schema when the client was implemented. Thus, dealing with extensions can be done in two ways:

- *Ignoring Unknown Data (weak forward compatibility):* The simplest strategy to deal with unknown extensions to the known data format is to ignore them. This (1) is only possible if they can be identified (which can be easily achieved with XML), and it (2) should only be done if ignoring extensions does not affect the semantics of the interpreted data.
  Well-known technologies using this kind of strategy are the rules for interpreting HTML's `object` element (rendering the object *or* the element's content), and SOAP's `mustUnderstand` attribute (flagging mandatory header blocks).

– *Interpreting Self-describing Data (strong forward compatibility):* Since ignoring unknown data often is not good enough (and only a very primitive way of dealing with extensions), a more interesting and promising way of reasonably implementing forward compatibility is to interpret extension data that is self-describing[1]. This strategy requires a framework for semantically describing data, and this question is discussed in Section 4.1.

Basically, forward compatibility is about "graceful degradation". Looking at the two extremes, one strategy is to ignore all data belonging to unknown extensions, a behavior which in many cases will not be accepted as being "graceful". The other extreme is that the self-describing data can be interpreted fully, without any compromises regarding the extension semantics. In this case, the new extension can be perfectly handled by the older application, and in essence, no application update is required, at least not from the extensibility point of view. In reality, however, many extensions fall between these two extremes, they are not fully ignored, but they are also not fully supported by earlier application versions. This difference between the intended behavior when dealing with an extension and the "graceful degradation" of older application versions is discussed in Section 4.3.

3 Schema Design Issues

Even though WSDL in principle can be based on any schema language, in practice in uses *XML Schema* [7] as its schema language. In its `types` section, WSDL defines the types that are used for exchanging XML messages. WSDL does not have an inherent way of versioning (nor do SOAP or UDDI), and the question so far has not been subject of any research activities. Thus, versioning WSDL descriptions is entirely within a user's responsibility, as is the corresponding management of schema information.

The core question of versioning Web Services is that of versioning XML Schemas. The two important issues are to design *open* and *extensible* schemas, which then can be used as a foundation for a Web Service extension framework. These two issues are defined as follows:

– *Openness:* An open schema allows unknown data to appear at certain points within an instance, so that instances of the schema may use extensions' instance data, but are still valid instances of the original schema.
  Since the goal of Web Service evolution is full compatibility, including forward compatibility, it is necessary to design the schemas in a way so that instances of extensions are also valid instances of earlier versions. To make this possible, a schema must be *open*, which means that it allows content in certain places without knowing the exact schema of the content. In basic XML, this can only be achieved through the DTD's `ANY` content specifier[2].
  XML Schema provides *Wildcards* for making a schema open. Using XML Schema's `any` and `anyAttribute` elements, it is possible to open a schema for unknown elements and attributes. Wildcards can be restricted to allow elements or attributes from certain namespaces only, which is discussed at the end of this section. The `processContents` attribute of wildcards can be used to specify whether wildcards should be validated; for the Web Service evolution scenario it should be set to `lax`, requiring validation if and only if a schema is available.
– *Extensibility:* An extensible schema is designed with versioning in mind, so that future revisions of the schema have clear ways of extending the schema. Versioning is built into the schema, and the schema's documentation makes clear how versioning should be implemented.
  Basically, there are two different issues when looking at extensibility. The semantics of an extension are unknown in advance, and have to be described using the declarative semantics described in Section 4. The schema of the extension, however, has to be described in XML Schema, and through its type layer, extensibility can be achieved in different ways.

[1] XML itself is often said to be "self-describing", but this is only true syntactically, since XML does not carry any semantics. If XML data should be able to convey self-describing semantics, an additional framework for this kind of information is required.

[2] However, `ANY` does not allow real openness, because it only allows any *declared* element.

Using type substitution, an instance may substitute the type of an element with a type derived from the original element's type. This substituted type must be declared in the instance with an `xsi:type` attribute. This mechanism can be used to use a schema in a more variable way than explicitly allowed by the schema. A limitation of this mechanism is that only elements may be substituted.

Using the `substitutionGroup` attribute of an element declaration, an element may declare itself as a member of the substitution group of another element. In this case, the substitution group member may be used in all places where the substitution group head is allowed[3]. Basically, substitution groups achieve an effect very similar to type substitution, but they are explicitly declared in the schema. Since substitution groups operate on element declarations, only elements may be substituted.

Openness and extensibility are issues that must be kept in mind when designing the first version of a schema, otherwise it will become unnecessarily hard to implement versioning. Through the `block` and `final` attributes of element and type declarations, it is possible to control type substitution and substitution groups. It is important to notice that XML Schema's default is to allow everything, which should be avoided by using the `blockDefault` and `finalDefault` attributes for the `schema` element and thus disallow everything, except for cases where these mechanisms are explicitly required and appropriate support is implemented.

As a side note, it has to be mentioned that much of the reuse of XML Schema is based on types, and the consequence of this is that processing of XML Schema instances should be type-based rather than name-based[4].

One last important consideration for XML Schema design is the use of *XML Namespaces* [8]. Namespaces can be structured in various ways, in some cases only containing XML element and attribute names, in other cases also containing additional subsets of names (such as for XML Schema, where a schema's namespace also contains type and other names). Namespace names must be syntactically correct URIs, but there is no need to actually associate a resource with a namespace's URI. However, it is useful to associate some descriptive resource with a namespace's URI, and a reasonable candidate for this is the *Resource Directory Description Language (RDDL)* or another language combining human-readable with machine-readable information.

The most interesting question when using XML Namespaces in a versioning scenario is whether different versions should use the same or different namespace names. There is no standard mechanism for namespace versioning, so the standard itself provides no support for declaring that an extension's namespace extends another namespace. However, when using a namespace description facility such as RDDL or something similar, it is reasonable and easily possible to include namespace dependency information in a namespace's description. In this case, using different namespace names for schema extensions are the preferable alternative. Whatever the namespace management approach may be (single namespace or namespace versioning), it is crucial to integrate this approach with the schema's wildcard declarations.

4 Extension Semantics

When discussing the semantics of schema extensions, two different forms of semantics must be clearly separated [9]. The first form of semantics are the *declarative semantics* that are hardwired into the extension framework through a declarative language, and are believed to be sufficiently expressive to cover the major issues of extensions. The second form are the *non-declarative semantics* (often also referred to as *procedural semantics*), which cover all aspects of extensions' semantics that cannot be captured by the declarative semantics.

[3] When using substitution groups, XML Schema's *identity constraints* become problematic, because they are based on element names, while the appropriate handling of substitution groups requires access to the type information. In some cases, this can be circumvented by using cleverly designed XPaths for the identity constraints, but in general substitution groups and identity constraints do not mix very well and should not be used in the same scope of a schema.

[4] Unfortunately, accessing type information is not properly supported through many XML APIs.

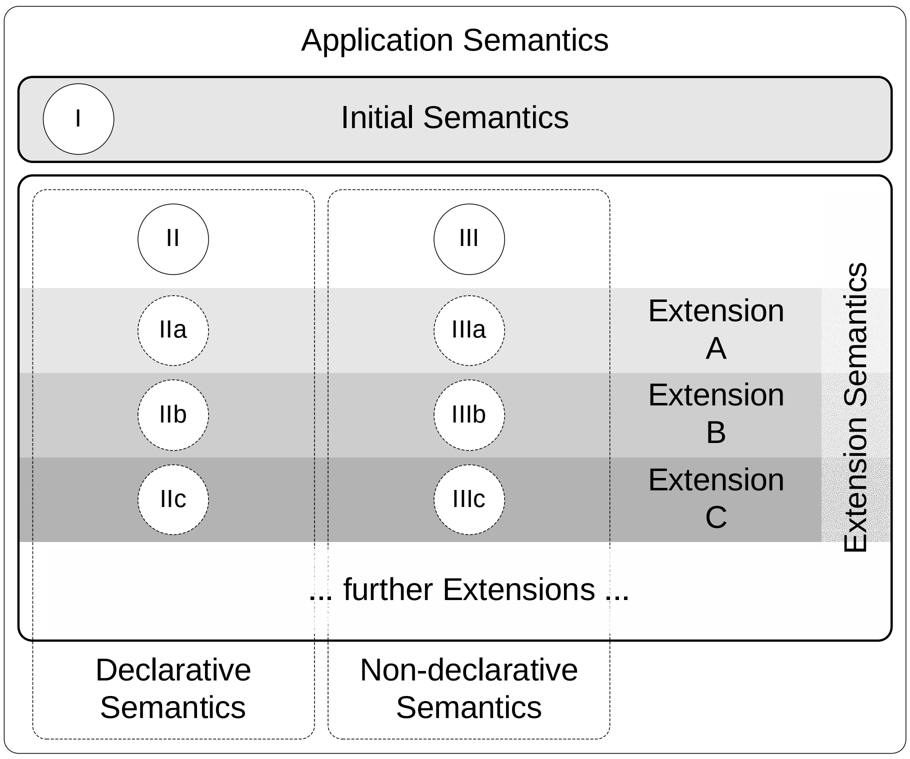

Fig. 1. Extension Semantics

Figure 1 shows the different forms of extension semantics. It also shows an application's *initial semantics* (I), which are the semantics of the application's initial vocabulary. Without semantic extensibility, an application is always limited to these semantics when dealing with unknown extensions. Through the usage of *declarative semantics* (II), however, it is possible to declaratively extend the semantics supported by an application to previously unknown extensions (IIa, IIb, and IIc). These declarative semantics are discussed in Section 4.1.

The second form of semantics are *non-declarative semantics* (III), which are the complement of the declarative semantics and thus together with the latter cover the "true" or "full" semantics of extensions. These semantics are discussed in Section 4.2. In order to find the right balance between expressiveness and complexity of declarative semantics, it is important to understand the difference between the two forms of extension semantics, and to deal with this difference in a reasonable way, so that there is the optimal balance between the extensibility of the schema, and the declarative expressiveness of the extensions' semantics. This question is discussed in Section 4.3.

4.1 Declarative Semantics

Since the goal is to make schemas semantically extensible (in addition to the simple vocabulary extensions discussed in Section 3), there must be a framework for describing semantics. This framework can range from extremely simple labels (e.g., `mustUnderstand` and/or `mayIgnore` labels) to very complex semantics, defining aspects of the extensions that may be used for different processing steps. This paper does not discuss the details of declarative semantics design, but instead focuses on the management and distribution of this information.

However, it should be kept in mind that semantics describe the meaning of extensions, and the meaning depends on the intended application (there is no meaning without context). This means that there can be different semantics for different facets or processing steps of the application scenario, which then are reflected in multiple semantics declarations for extensions.

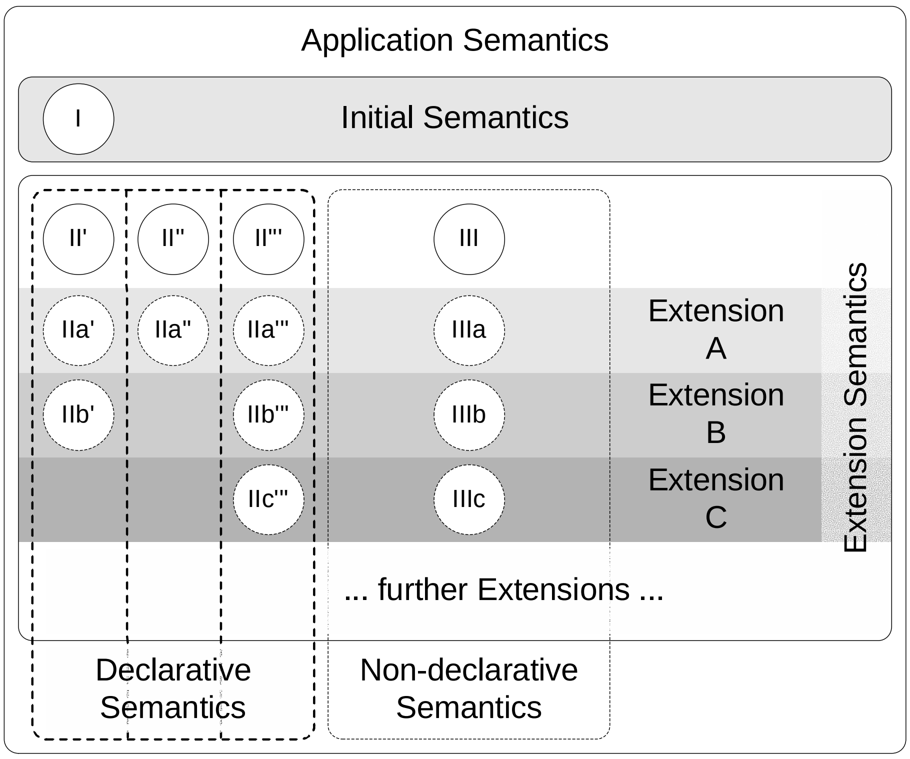

Fig. 2. Multiple Declarative Semantics

Figure 2 shows an example of multiple declarative semantics (II′, II″, and II″′). It is also shown that not every extension must use all declarative semantics. If in the application context it is sufficient for an extension to be described using a subset of the available declarative semantics, then this is perfectly acceptable. Any extension using only a subset of the available declarative semantics then can only be interpreted in the context of these particular semantics.

Declarative semantics are necessarily represented in some form of formalism, and the ability to interpret this formalism is required for all components of the extension framework. Without this ability, extensions can not be described semantically[5]. In case of the extension framework presented here, there is the question of whether to encode the semantic information within instance data (in which case we speak of *extensional semantics*), or within the schema (called *intensional semantics*):

– *Extensional Semantics:* In this case, the semantics declarations are part of the instance data[6], so that each instance is completely self-describing, at least with respect to the declarative semantics. The advantage of extensional semantics is the ability to access the semantics by interpreting the instance, rather than having to access the schema. However, there is a large overhead associated with this, in particular if the semantics declarations are non-trivial. Another drawback is that the application has to trust the instance originator (rather than the schema originator) to describe the semantics, which can be a severe security issue.
– *Intensional Semantics:* Since extensions are described in the schema (rather than individually by instance authors), it makes more sense to describe their semantics in the schema, too. In

[5] Of course, if an implementation does not need to understand all kinds of declarative semantics, if there is more than one, it is possible to have implementations supporting only a subset of the existing declarative semantics.

[6] Naturally, the schema must define some mechanism to represent this information.

"Extensional" HTML Extension (Version 1):

```
<div style=" [[ CSS Declarations ]] "> ... </div>
```

"Extensional" HTML Extension (Version 2):

```
<style type="text/css">.textbox { [[ CSS Declarations ]] }</style>
  ...
<div class="textbox"> ... </div>
```

Borderline HTML Extension (depending on the `textbox.css` ***origin):***

```
<link rel="stylesheet" type="text/css" href="...textbox.css" />
  ...
<div class="textbox"> ... </div>
```

"Intensional" HTML Extension:

```
<textbox> ... </textbox>
```

Fig. 3. Extensional vs. Intensional HTML Extensions

this case, extensions in an instance cannot be interpreted semantically without accessing the extension schema, which is the drawback of the intensional approach.

There are drawbacks and advantages for both type of declarative semantics, but for most scenarios, the intensional approach is preferable, because it avoids the duplication of semantics declarations in instances, and because it avoids some of the security issues of the extensional approach. Furthermore, the extensional approach is best suited for cases where instance originators need to make up ad-hoc extensions for individual instances, which in most application scenarios is not a requirement.

However, there is one important exception to the general observation that intensional semantics are preferable over extensional semantics, and this is the case when semantics absolutely need to be observed, even in cases where the application can or does not want to access the intensional semantics. The most popular example for this kind of semantics is the `mustUnderstand` semantics, which signals to the application that it must be able to process the semantics of this extension[7]. Since this kind of information is crucial to the flawless function of the extension framework, it must be represented by extensional semantics.

In order to demonstrate these types of extensions, it is interesting to look at HTML, which supports an interesting combination of extension mechanisms. HTML provides hooks for embedding semantics (in case of HTML, these are formatting semantics) into an instance, but also makes it possible to describe the semantics of extensions in separate resources, which are referenced by instances. Since it is not quite clear whether these separate resources should be regarded as an external part of the instance or of the extension schema (if there is one)[8], this case is labeled as "borderline" in Figure 3, which shows the possible combinations of HTML's mechanisms.

[7] Whether the declarative semantics are sufficient or not depends on the application. Consequently, in many cases it probably makes sense to differentiate between `mustUnderstandDeclaratively` and `mustUnderstandFully`.

[8] The most reasonable way to decide this is probably the author of the external resource. If the external resource has been provided by the schema author, then the instance uses intensional semantics, and extensional semantics otherwise.

The example for extensional vs. intensional extensions shown in Figure 3 assumes (for demonstration purposes only) that a new element `textbox` should be introduced in HTML for separate text boxes (in fact, flowing "textual figures" just like Figure 3 itself). For the first extensional case, the extension is implemented as an existing HTML element (HTML's `div` element, which does not have any inherent formatting semantics) and directly associated style information. The style information uses CSS, which is the declarative formatting semantics language for HTML. There is no guarantee that every intended extension to HTML can be described through CSS declarations, but the CSS vocabulary covers a broad area of formatting semantics, so that many extensions can in fact be described using CSS. Version 2 of the extensional case shows a variation, using the `class` mechanism supported by HTML and CSS. From the semantics point of view, Version 2 is just an abbreviation of Version 1, which enables the reuse of style information inside an HTML page.

The borderline version also uses the `class` mechanism, but the HTML class refers to an external CSS resource. As mentioned above, this can be regarded as extensional or intensional, depending on the author of the external CSS resource.

In the intensional case of the example in Figure 3, the new element `textbox` simply appears in the HTML (which should use a `DOCTYPE` declaration declaring the new version), and it is assumed that semantic information about the formatting of this new element can be found in the extended schema, or, in case of HTML and CSS, in an associated style sheet (generated from the schema or authored separately), where there can be CSS code associated with a `textbox` element type selector, containing the same code as the `.textbox` class selector of the extensional case. Today's browsers are not equipped to dynamically adapt to new HTML versions, but most hooks are there to implement such a behavior.

To summarize, HTML provides support for extensional as well as for intensional ways to declare the semantics of extensions. However, since there is no standardized way to associate an "extension formatting semantics CSS" with a new version of HTML (instances as well as schemas), all browsers available today do not support schema-based extensions; all that can be implemented through the browser mechanisms are instance-based extensions. The three main reasons for this lack of dynamism in modern browsers are (1) that HTML has remained stable for a long time now, (2) the lack of a standardized way of communicating non-declarative semantics, and (3) the browser authors' assumption that extensions of HTML probably not only require declarative extensions via CSS, but also processing semantics beyond the capabilities of CSS, which require a browser update to be fully supported.

4.2 Non-declarative Semantics

The approach of declarative semantics discussed in the previous section is to describe as much of the foreseeable semantics of extensions as possible. This makes it possible to make extensions semantically self-describing, at the price of having to implement the declarative semantics[9]. Generally speaking, declarative semantics in almost all cases will only cover a part of the full semantics intended for extensions, and these "full" or "true" semantics of extensions are called the *extension semantics*. In Figure 1, the declarative semantics (II) and the non-declarative semantics (III) together constitute the full extension semantics.

The most important question for the design and usage of declarative and non-declarative semantics is whether the mismatch between these two can be tolerated for a useful fraction of application scenarios. This semantics mismatch is discussed in more detail in Section 4.3. It should be noted that, in contrast to the purely declarative semantics, which by definition can be fully described through declarative mechanisms, the non-declarative semantics are often defined less formally, either by prose, or through procedural code performing certain functions.

In order to give an example for non-declarative semantics, Figure 4 shows an hypothetical extension of HTML with an element for marking up square root content, which should be formatted as a square root formula.

[9] In the case of CSS as described in the previous section, this price is quite high. CSS has become so complex that there is no single browser supporting the full CSS language repertoire.

HTML Extension for Square Root Element:

```
Square root rendering looks like <sqrt>2</sqrt>
```

Declarative Formatting Semantics (CSS):

```
sqrt:before { content : "sqrt(" }
sqrt:after  { content : ")" }
```

Formatting of HTML Extension:

Declarative Semantics:	Square root rendering looks like sqrt(2)
Non-declarative Semantics:	Square root rendering looks like $\sqrt{2}$

Fig. 4. Declarative vs. Non-declarative Semantics for HTML Formatting

Assuming that the declarative semantics of the HTML extension is CSS and implemented intensionally, one possible way to describe the extension declaratively is to simply insert the text `sqrt(` before and the text `)` after the content of the `sqrt` element. This is not a particularly elegant way to render a square root, but is sufficient to convey the meaning of the content, and therefore probably acceptable.

However, the “full” semantics of the extension are to render the content as a square root, and the standard way to do this is to use a square root symbol ($\sqrt{}$) and to include the content under the top bar. Therefore, an extended client implementing the extension (or, in other words, implementing the non-declarative semantics of the extension) can format the square root using the appropriate mathematical symbol.

In this case, the difference between the declarative semantics and the non-declarative semantics is obvious. However, most users probably agree that rendering the square root as `sqrt()` is good enough to create an understandable rendering of the markup, so the difference in semantics is acceptable. This question is discussed more generally in Section 4.3. In this case it is also interesting to note that the non-declarative semantics effectively “overwrite” the declarative semantics. This happens frequently, but the overlap between declarative and non-declarative semantics varies and cannot be described generally.

As mentioned in the introduction to this section, semantics always depend on context. Consequently, non-declarative semantics often include new and unforeseeable context (such as new processing steps) that cannot be covered by the declarative semantics of a given extension framework. The interesting question always is whether it is possible to describe extensions sufficiently using the declarative semantics allowing a “graceful degradation” of the behavior as described in Section 2.

4.3 Semantics Mismatch

Starting from Figure 1, it can be asked whether the initial semantics of the application (I) are or at least can be described using declarative semantics. Usually, this is not the case, because of the unnecessary effort this requires, and also because the initial semantics often are more complex than what can be described with the rather limited declarative semantics for extensions.

Applications interpreting data with unknown extensions are limited to the declarative semantics (II) contained in the instance (extensional) or the extension schema (intensional). In most cases, the full extension semantics are a superset of the declarative semantics, also described by non-declarative semantics (III). The non-declarative semantics, however, are implemented only by applications having advance knowledge of the extension, so earlier versions of the application are limited to the declarative semantics. These limited semantics can have two consequences:

- *Graceful Degradation:* If the limited declarative semantics are sufficient to guarantee the correct functioning of the application, possibly with some compromises regarding the quality of extension processing, then it is reasonable to accept the limitation in semantics. The earlier example from Figure 4 of rendering a square root textually is an example of how the limitation of the declarative semantics is acceptable, because the limited extension processing still yields the intended result: a human-readable form of displaying the square root of an operand.
- *Extensibility Failure:* If it is considered inadequate to process an extension or particular instances of an extension with declarative semantics only, then this should be signaled to older applications by specifying this information, ideally as extensional information (i.e., as part of the instance data). The application behavior regarding extensibility failures can range from signaling an error to more graceful behavior such as recommending updating the application to enable better processing of this particular extension respectively its instance data.
  Moreover, extensibility failures can raise different error levels, and they can be specific to a type of declarative semantics, so that only applications interpreting these semantics are affected by the extensibility error. Consequently, extensibility failures should be more specific than a simple `mustUnderstand` flag, which is the simplest form of extensibility failure information. Most likely, extensibility failure information should include the error level, and the affected semantics.

Generally speaking, the question of how well the declarative semantics can handle the actual evolution of the application depend on the complexity of the declarative semantics, and of course the ability to design these semantics in advance. An interesting example is the ongoing evolution of CSS. CSS can be considered the declarative semantics (for formatting information) of HTML, and the specification has grown from a rather limited set of formatting instructions in CSS1 to the complex variety of formatting modules constituted by CSS3 [10]. The problem is that CSS has become so complex that no browser fully supports the complete CSS vocabulary, which is an indication that the costs of implementing the complex declarative semantics seem to outweigh the benefits.

During CSS evolution, new semantics were added to CSS, for example the ability to specify table formatting, which was not included in earlier versions. However, even CSS3 does not cover all areas of HTML or possible extensions to HTML, for example there is no support for hyperlinking or mathematical formulas. Nonetheless, CSS is an interesting example of an *extension of extension semantics*. The price for this is that older versions of browsers do only support earlier versions of CSS, so that they cannot interpret all CSS information that may be used in newer documents. In case of HTML/CSS, the standard mechanism is to ignore unknown instance data[10], but in the general case, the management of the initial schema, extensions to it, declarative semantics, and the extension of declarative semantics is a challenging task.

Again, it is a question of how much effort one would like to invest into the design and implementation of the extension framework, and how much benefit this effort and cost is going to generate. Because the *extension of extension semantics* introduces a new level of complexity, it is not discussed in this paper.

5 Extension Framework

The complete framework for semantically extensible schemas for Web Service evolution combines the aspects presented in previous sections with guidelines and rules to control the evolution and to mandate processing guidelines which must be observed by peers participating in the Web Service scenario. The framework includes:

[10] Ignoring unknown instance data is the required processing behavior for HTML elements and attributes, and for CSS selectors and declarations.

– *Open and Extensible Schemas:* Without appropriate schema modeling, there can be no controlled evolution, so the design guidelines for openness and extensibility of schemas outlined in Section 3 must be followed.
– *Extension Semantics:* Since the goal is the semantic extensibility of Web Services, the rules for extension semantics as presented in Section 4 must be followed. Without extension semantics, extensions can only be ignored.
– *Extension Guidelines:* Since current Web Service technologies (SOAP, WSDL, and UDDI) do not provide any support for versioning, the versioning process must be controlled by the user. This should be done through a set of extension guidelines, which document all necessary steps and interdependencies that are required for defining, implementing, and deploying a new version of a Web Service.
– *Processing Rules:* Participating peers must follow certain guidelines when implementing extensible Web Services, in particular they may not ignore information that is required for proper processing of extensions. Since these rules go beyond basic Web Service semantics, they must be enforced by mandatory processing guidelines for participating peers.

Implementing this framework in a Web Service scenario is not trivial. In particular, the design of declarative extension semantics (possibly including multiple semantics for different application aspects) can be hard. Furthermore, the extension guidelines impose additional constraints on schema designers, and the processing rules impose additional constraints on implementors. It is thus necessary to carefully weigh the cost of the extension framework against the benefits of using it. Important factors in this analysis include:

– *Controllability of Software Versions:* If the application scenario is a closed environment where software updates can be controlled and can be realistically deployed within a reasonable time frame, then investing in software version management may be a better solution.
– *Server/Client Ratio:* The hard part in extensibility is forward extensibility, which in case of Client/Server-Scenarios means old clients with new servers. If the number of clients is small, client updates can be achieved much easier. If the number of servers is small, backward compatibility may not be an issue, because servers can easily be updated.
– *Frequency and Locality of Extensions:* Building extensibility into the software is only reasonable if the frequency of extensions is higher than the normal versioning cycle of the software. Frequent extensions, or extensions being introduced by different participants in the application scenario make the effort of implementing extensibility more likely to be beneficial.
– *Predictability of Extension Semantics:* They key of useful extension semantics is the ability to capture as much of the semantics of potential extensions as possible, because only then it is possible to design a declarative language for them. If the application scenario is likely to produce extensions with significant semantics that cannot be represented declaratively, then either the declarative extensibility approach is inappropriate, or it should be upgraded to an (even more complex) approach for extensible extension semantics.

Thus the framework for semantically extensible schemas for Web Service evolution is not the perfect solution for every Web Service scenario. However, there are many scenarios which may benefit from a managed way of Web Service versioning, and declarative extension semantics provide a particularly powerful way of managing extensions. The framework presented in this paper is not formally complete, and has evolved from concrete application scenarios rather than a purely top-down approach. We therefore think that further work is required in this area, and that more real-life applications are required to find the optimal balance between framework complexity and the average cost/benefit ratio.

6 Conclusions

The framework presented in this paper is the result of two real-life scenarios where extensibility was a key point of the requirements. However, before it can be turned into a completely defined formal way of designing semantically extensible Web Services, more examples are needed for evaluating the framework.

Also, some rather complex problems have only been discussed very shortly, such as the question of if and how the extensions of the schema (discussed in Section 3) and the associated semantics (Section 4) can and should be coordinated in a well-defined and robust way. Then there is the interesting issue of an evolution of the declarative semantics, so that there can be a need for an extension framework for these semantics (which thus addresses the issue of *extensible extensibility*).

Since wide-scale Web Service deployment still is a rather rare case (in most cases, Web Services are used in small and controlled settings), the question of Web Service versioning so far has not become a real problem. In the medium term, however, this will happen, and the key Web Services specifications (SOAP, WSDL, and UDDI) will need to provide better and more advanced answers than SOAP's current `mustUnderstand` attribute, the only means of versioning support in the current Web Service specifications. This paper gives no final and perfect answers, but presents approaches which provide a reasonable solution to the problem of Web Service evolution.

Once Web Services are deployed more widely and more loosely than today, Web Service evolution will become a serious problem. Current Web Service technology does not provide adequate support for this type of problem, and the framework for semantically extensible schemas for Web Service evolution presented in this paper augments the existing Web Service technologies to better support Web Service versioning.

References

1. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Third Edition). World Wide Web Consortium, Recommendation REC-xml-20040204 (2004)
2. Chinnici, R., Gudgin, M., Moreau, J.J., Schlimmer, J.C., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. World Wide Web Consortium, Working Draft WD-wsdl20-20040326 (2004)
3. Orchard, D., Walsh, N.: Versioning XML Languages. World Wide Web Consortium, Proposed TAG Finding (2003)
4. Su, X., Brasethvik, T., Hakkarainen, S.: Ontology Mapping through Analysis of Model Extension. In Eder, J., Welzer, T., eds.: Short Paper Proceedings of the 15th Conference on Advanced Information Systems Engineering. Volume 74 of CEUR Workshop Proceedings., Klagenfurt, Austria, Technical University of Aachen (RWTH) (2003) 101–104
5. Hunter, J.: MetaNet — A Metadata Term Thesaurus to Enable Semantic Interoperability Between Metadata Domains. Journal of Digital Information **1** (2001)
6. Castano, S., Ferrara, A., Kuruvilla Ottathycal, G.S., De Antonellis, V.: A Disciplined Approach for the Integration of Heterogeneous XML Datasources. In: Proceedings of the 13th International Workhop on Database and Expert Systems Applications (DEXA 2002), Aix-en-Provence, France, IEEE Computer Society Press (2002) 103–110
7. Fallside, D.C., Walmsley, P.: XML Schema Part 0: Primer Second Edition. World Wide Web Consortium, Proposed Edited Recommendation PER-xmlschema-0-20040318 (2004)
8. Bray, T., Hollander, D., Layman, A., Tobin, R.: Namespaces in XML 1.1. World Wide Web Consortium, Recommendation REC-xml-names11-20040204 (2004)
9. Ghezzi, C., Jazayeri, M.: Programming Language Concepts. 3rd edn. John Wiley & Sons, Chichester, England (1997)
10. Meyer, E.A., Bos, B.: CSS3 Introduction. World Wide Web Consortium, Working Draft WD-css3-roadmap-20010523 (2001)