UNIVERSITY OF ECONOMICS, PRAGUE

Compound XML documents validation

Diploma thesis

Petr Nálevka

Supervisor: Ing. Jiří Kosek

April 2007

Annotation

This thesis investigates different aspects of compound XML documents and shows the potential benefits of using such documents in today's Web environment. The major concern is dedicated to issues related to compound document validation. Possible approaches to address those issues are examined. One of the validation methods called NVDL (Namespace-based Validation Dispatching Language) is discussed in detail. This thesis describes the main principles of NVDL, analyses the advantages and disadvantages over other similar concepts and introduces JNVDL. JNVDL is a complete Java-based implementation of the NVDL specification developed as part of this thesis. The implementation is described from low-level technical perspective as well as from user-level perspective. As a proof of the concept, JNVDL has been integrated into an existing Web documents validation project called Relaxed to make straightforward compound document validation available to Web content authors.

The first chapter of this thesis explains what are compound documents and why should they be used. This chapter shows examples and technical background of compound documents. Special attention is paid to compound document validation issues. Different approaches are compared and their advantages and disadvantages are highlighted.

The whole second chapter is dedicated to the NVDL specification. NVDL is a standardized approach for compound document validation. This chapter contains detailed description of NVDL language semantics and the NVDL validation dispatching process and it can serve as an tutorial for those who would like to learn the language.

The third chapter describes JNVDL; an NVDL specification implementation written in Java programming language. This chapter shows the implementation details as well as the usage of JNVDL. It is a long way for a technology to get from a specification stage to real world applications. The third chapter describes, how did JNVDL coped with different issues not addressed in the specification.

JNVDL in use is presented in the last chapter. This chapter shows integration of JNVDL into an existing Web documents validation project called Relaxed.

Anotace

Tato práce se zabývá různými charakteristikami komponovaných dokumentů a ukazuje potencionální výhody využití takových dokumentů v prostředí dnešního Webu. Hlavní pozornost je soustředěna na problémy spojené s validací komponovaných dokumentů. Práce zkoumá různé přístupy k řešení těchto problémů. Validační metoda NVDL (Namespace-based Validation Dispatching Language) je popsána detailně. Tato práce popisuje hlavní principy NVDL, zkoumá výhody a nevýhody oproti jiným přístupům a představuje JNVDL. JNVDL je kompletní implementace specifikace NVDL, která byla napsána v jazyce Java jako součást této práce. Popsány jsou nejen technické prvky implementace, ale JNVDL je představeno i z uživatelské perspektivy. Pro ověření využitelnosti bylo JNVDL integrováno do existujícího projektu pro validaci webových dokumentů s názvem Relaxed, aby jednoduše zpřístupnilo validaci komponovaných dokumentů autorům webového obsahu.

První kapitola této práce se zabývá komponovanými dokumenty a důvody pro jejich použití. Tato kapitola ukazuje, jak jsou komponované dokumenty řešeny technologicky a také příklady takových dokumentů. Zvláštní pozornost je věnována problematice validace komponovaných dokumentů. Různé přístupy jsou porovnávány s důrazem na jejich výhody a nevýhody.

Celá druhá kapitola je věnována specifikaci NVDL. NVDL je standardní přístup pro validaci komponovaných dokumentů. Tato kapitola obsahuje detailní popis sémantiky jazyka NVDL a validačního procesu, který popisuje. Druhá kapitola může být také užitečná pro ty, kteří se chtějí naučit používat tento jazyk.

Třetí kapitola popisuje JNVDL, což je implementace specifikace NVDL napsaná v jazyce Java. Tato kapitola popisuje nejen implementační detaily, ale i užití JNVDL. Od specifikace k zavedení technologie do praxe vede dlouhá cesta. Třetí kapitola se proto zmiňuje o tom, jak si JNVDL poradilo s různými problémy, které specifikace explicitně neřeší.

Praktické nasazení JNVDL je demonstrováno v poslední kapitole. Tato kapitola ukazuje integraci JNVDL do existujícího projektu pro validaci webových dokumentů, který se jmenuje Relaxed.

Declaration

This thesis was elaborated individually by myself with the application of literature mentioned in the attached list of references. I have no objections to lend the thesis with the agreement of the faculty department or to publish the thesis or a part of it.

28th of April 2007, Prague

Petr Nálevka

Table of Contents

Preface

Over the last years XML become very popular. It features a simple format which is easily processed by machines and human readable at the same time. There are many techniques which help developers to work with XML. XML documents can be seamlessly transformed using transformation languages, presented using styling and templating languages and their grammar may be described and validated using schema languages. Most programming environments on various platforms have some XML API allowing developers to manipulate XML documents. That's the reason why today, XML is widely adopted in many areas, on the Web as well as in the enterprise. XML is used as an universal data exchange and storage format especially useful in heterogeneous environments. It is also frequently used as a format for complex configuration. Today there are many different XML languages designed to solve different problems.

At first, those languages has been used separately, but as XML matures, it is more and more obvious that some problems may be solved smarter using a combination of more different single-purposes languages rather than using one complex versatile language. If there already exists a well adopted and understood language which solves part of a problem, it makes a good sense to reuse it rather than introducing something new.

The designers of XML obviously considered such concerns as they introduced XML Namespaces; a flexible way to combine different mark-up vocabularies[1] in a single XML document. The mechanism of XML Namespaces addresses the problem of collision and recognition of elements and attributes from different vocabularies.

With XML Namespaces, it is technically no issue to create compound XML documents, but an important issue considered in this thesis is validation of such documents. Validation is an essential part of every XML manipulation process. There are basically two levels of document instance correctness. First, the document instance is well-formed if it adheres to the general XML syntax. Only well-formed documents may be processed using XML parsers. Second, the document instance is valid, if and only if it is well-formed and if it satisfies its language schema[2] definition in every aspect.

A validation process basically matches given document instance with its schema. The reason why validation is so important is apparently interoperability. Where well-formness ensures the document can be even parsed, validity ensures the document

[1] The term vocabulary, or mark-up vocabulary represents a set of elements and attributes based in the same namespace.

[2] Schema is a description of the structure or rules an XML document must satisfy. It includes the formal declaration of elements and attributes that make up a language.

can be fully "understood" by a different information system or application. Validity violation often causes interoperability problems. To ensure correct processing of XML documents, it is essential to first check their validity.

When using a single-vocabulary document, validation is a straightforward task. The whole document is validated against one schema. The situation gets more complicated for compound documents as it is necessary to validate different language fragments against different schemas and those are often defined using different schema languages.

Another issue stems from the fact there are many different possibilities to combine two XML vocabularies, but not all of them are desirable. Sometimes it makes sense to allow foreign grammar just in an eligible context and forbid it elsewhere.

This thesis shows how to overcome those difficulties using different aproaches to compound document validation. One of the approaches which is discussed in great detail is NVDL (Namespace-based validation and dispatching language) which is a brand new international standard for compound document validation. A Java-based implementation of NVDL specification called JNVDL was developed within the scope of this thesis. This implementation is further closely described together with the general principles of NVDL. This thesis also demonstrates a real application of JNVDL, which was integrated into an existing Web document validation system to extend its compound document validation capabilities.

Chapter 1

Compound documents

It is almost eight years ago when W3C introduced the Namespaces in XML recommendation, which made it technically feasible to combine different XML vocabularies in one document[1]. For some people, namespaces where a hostile element polluting XML with additional complexity without any reasonable need for it. Despite initial problems, advantages of namespaces become more and more obvious and today they are used in many different applications.

Web and its history is a good example to demonstrate, why combining vocabularies is the right approach to solve some sort of language extension problems. At first, HTML was a simple format for publishing mostly scientific articles and for linking them together. But soon, Web became incredibly popular and its wide and fast adoption astonished everybody. It became not just a standard for publishing electronic documents in general, but it also become a standard way to create distributed user interfaces to make all sort of information systems accessible.

The tendency to use or misuse HTML to solve all sorts of distinct problems slowly polluted the language with various indiscreet proprietary extensions. For example presentational aspects have been mixed up with semantical aspects. The situation got even worse as there was almost no standardization. Browser and other software vendors were promptly introducing new user-oriented features to gain competitive advantage without having a long term vision of the language development in mind. All those issues slowly caused the Web was loosing one of its main advantage; cross-platform interoperability. HTML simply become a big monolithic language intended to solve all sort of problems, but not solving any of them in a satisfactory manner.

The situation has improved when the language became standardized under W3C and the main vendors agreed on the standards to some extend. Today, the language is slowly improving and evolving. A big issue slowing things down is of course backward compatibility, but over the last years, the language has been purified from most of the presentational aspects[2]. Moreover, HTML has been made fully XML based (XHTML) and it has been modularized. Modularization allows to use exclusively those features which are really needed for a particular application. Related aspects of the

[1] Today, such documents are called compound documents.
[2] Presentation of HTML documents shall be expressed through stylesheet languages.

language are decomposed into various modules; e. g. a structure module with the main HTML structure, a table module for expressing tabular data, text module for organizing text into headings and paragraphs, forms module to build form related interaction with users, meta-information module to attach meta-data to documents, image module for images, link module to hyperlink documents.

A different set of XHTML modules can be used for addressing different problems. For example, it doesn't make sense to use the forms module in case of publishing a simple electronic document using XHTML. Moreover, modules can be understood as simple default vocabulary fragments with limited functionality. They can be left out completely, if there is no need for them, or they can also be replaced by a whole different specialized and feature-rich language. This can be clearly spotted as the current trend in Web standards. Instead of the forms module, XForms can be used, RDF can replace the meta-information module, link module can be overridden by XLink, instead of images, SVG vector graphics can be directly embedded etc...

Such approach makes HTML and Web much more flexible and powerful. It makes it also more suitable for all sort of completely different applications. Specific problems can be addressed through a special combination of languages. Moreover, highly specialized single-purpose vocabularies can be reused even outside the Web environment for all sort of different tasks. This kind of thinking inevitably leads to the point, where no generic-purpose language as HTML is needed anymore. The only thing needed is a parent language to embed all the desirable vocabularies. The XHTML structure module is a good candidate, but a completely different language can serve this purpose as well. For example, full-featured Web applications can be created using SVG with embedded XForms. Where SVG serves as the parent language and it covers presentation, XForms are used to collect user data.

Of course there are many unresolved issues related to compound documents and some of them can be addressed technologically; e. g. validation problems which are discussed in this text. But despite those issues, compound documents bring significant advantages and a new level of flexibility, not just to the Web environment, but to many different areas.

1.1. Namespaces in XML

The “Namespaces in XML” recommendation [NS] addresses problems of recognition and collision of different vocabulary elements and attributes within one XML document. The idea behind XML namespaces is very simple and straightforward. Without namespaces, it would be necessary to assure none of the vocabularies used within one document has clashing names of elements and attributes. If they do, applications cannot “understand” those elements and attributes correctly as it is not obvious which vocabulary they do belong to.

Namespaces solved this issue by assigning extended names to elements and attributes. An extended name is a pair of a namespace name and a local name (local names

are defined in [XML], namespace names are represented by an URI which is specified in [RFC3986]). As URIs may contain characters not allowed in names and they also tend to be quite long, elements and attributes are rather bound to namespaces using qualified names. A qualified name can be prefixed or not. A prefix is nothing more than just a placeholder for the namespace. A prefix is mapped to a namespace using an `xmlns:prefix` attribute.

Example 1.1. Namespace declaration and prefixes

The `http://example.org/bibliography` namespace is declared at the `book` element. The namespace is represented by the `bib` prefix. All qualified names in the example are prefixed and they do belong to the `http://example.org/bibliography` namespace.

```
<bib:book xmlns:bib="http://example.org/bibliography">
  <bib:title>Effective XML</bib:title>
  <bib:subtitle>50 Specific Ways to Improve Your XML</bib:subtitle>
...
</bib:book>
```

Elements with unprefixed qualified names belong into the default namespace which is declared using the `xmlns` attribute. Unprefixed attributes are assigned to an empty namespace and their interpretation depends on the element where they occur. The default namespace construct allows to minimize the use of prefixes within documents making them shorter and more readable.

Example 1.2. Default namespace

The `http://example.org/bibliography` namespace is declared as the default namespace. Where the unprefixed elements `book` and `title` belong to the default namespace, the `include` element belongs to `http://www.w3.org/2001/XInclude`.

```
<book xmlns="http://example.org/bibliography" ▶
xmlns:xi="http://www.w3.org/2001/XInclude">
  <title>Effective XML</title>
  <xi:include href="introduction.xml"/>
</book>
```

The scope of a namespace declaration begins at the start tag where the declaration appears and ends at the corresponding end tag excluding any inner declaration scopes for the same namespace. For example, when declared at the root element of the document, the prefix's scope is the whole document, unless there is another namespace declaration for the same namespace at one of its descendants. The namespace declaration applies to every element and attribute with the matching prefix within the scope. Multiple prefixes may be declared at each element.

1.2. Compound document applications

Compound documents already have many different applications in many different areas. This section mentions some of the typical use-cases.

1.2.1. Templating languages

One of the first applications of compound documents were templating languages. The W3C XSL Transformations recommendation [XSLT] has been published just few months after Namespaces in XML. XSLT is a popular templating language. It can express rules for transforming a source XML tree into a result tree. The transformation is done by associating patterns with templates. Where patterns match parts of the source tree, templates create the corresponding result. XSLT is powerful enough to completely change the structure of the source tree in the result.

The particular rules are expressed using the XSLT language, which is an XML vocabulary assigned into the XSLT namespace. In case the result is again an XML document, fragments of the result tree need to be placed within the appropriate templates. As those fragments may belong to an arbitrary namespace, such XSLT stylesheet is in fact a compound document.

Another widely used templating language which features an XML notation is JSP (Java Server Pages). JSP is used as a standard scripting language for developing Web pages in Java. In JSPs, custom mark-up assigned to different namespaces is bound to Java code which is than executed during render time of the pages. Fragments of the resulting page (usually HTML fragments) can be embedded within such custom mark-up. To conclude, such JSP script is again a compound document.

1.2.2. XML protocols

XML is widely used as a data exchange format. In non-trivial cases, a protocol is needed to exchange structured XML messages. SOAP (Simple Object Access Protocol) can be mentioned as an example of such a protocol. In SOAP, XML queries are sent to retrieve XML responses. In general, those are called SOAP messages, which are basically XML data wrapped up in a SOAP envelope. The envelope, which consists of a header and a body, carries also message related meta-data such as processing information.

No doubt, SOAP messages are again compound documents as they consist of an envelope (SOAP vocabulary) which encapsulates the actual message payload (arbitrary vocabulary).

1.2.3. Office documents

Office documents are exactly the right example for a compound document application. It is actually the environment, where the phrase “compound documents” originated.

Office documents need to solve exactly the kind of problem which is addressed through namespaces in XML. There are various differently structured data (e. g. text, tables, graphs, diagrams or spreadsheets) and those need to be incorporated into one document.

XML is undoubtedly the right format to serialize office documents. The reason is, there are many specialized mature XML vocabularies ready to be reused as a format for describing different type o data traditionally used within office documents and XML features a good mechanism (namespaces) to combine those vocabularies together. Open Document Format (ODF) can be mentioned as an example of XML being successfully used in various office bundles as the native document format. ODF is again a compound document format.

1.2.4. The Web

As mentioned previously, there is a big opportunity for compound documents in the Web environment. There are many different standard languages which can significantly enhance the Web user experience and make Web documents more flexible and tailored to specific applications.

A nice example of a compound document concept designed for use in the Web environment is the xH language presented for example at the WWW2006 conference. It is basically a synthesis of XHTML and several other well-know standard XML languages; mainly XForms, SVG and MathML. Interaction of the different language blocks is achieved through JavaScript.

All the languages recommended to be used within xH are widely used and well known. But the value added and the major intention of xH is to encourage people to use those languages in combination to build a new generation of flexible Web applications with enhanced user experience.

1.3. Compound document validation

XML namespaces did technically allow presence of multiple vocabularies inside one XML document, but this was just the beginning. There are many other issues which need to be addressed before adopting a compound document solution.

Standalone XML languages usually have their semantics described verbally in the language specification. Such specification explains the meanings of different elements and attributes and defines the way they should be interpreted by different applications. The language syntax is usually formally defined in a schema. A well-designed schema reflects the semantics, as it allows to construct just meaningful structures with unambiguous interpretations.

When using compound documents, a completely new level of complexity emerges. In this case, different language fragments are combined within one XML document instance. Such fragments can be combined in many different ways and even the isolated

fragments are meaningful (in respect to their language semantics) and they are syntactically correct, the combination of such fragments can be difficult to interpret or it may be even semantically empty.

An XHTML document which consists of a `head` section and a `body` section can serve as an example. In contrast to the `head` section, which is used to place various meta-data related to the document, the `body` section is intended to be rendered. In case, it is desirable to attach some RDF meta-data to the document, placing such fragment into the `body` section would cause interpretation difficulties as meta-data are not intended to be rendered. The correct place to put RDF fragments is of course the `head` section.

This implies, there are two major issues concerning compound documents. First, semantics needs to be created for different compound languages[3], to make them correctly interpretable by different applications. Second, the way different languages are combined together has to be constrained to allow just the meaningful combinations (compound language syntax). This requires some kind of a "meta-schema" language to express such constraints.

Today, for standalone XML languages, automated validation is absolutely essential to ensure their interoperability. This is even more important for compound documents, as they tent to be more difficult to interpret. To make compound documents applicable in a heterogeneous environment (as the Web environment for example), it is absolutely necessary to provide powerful validation tools and techniques. This requires schema languages able to cope with multiple namespaces and validation engines able to check document instances against such schemas. Two distinct approaches to compound document validation are discuss in the following sections.

1.3.1. Current schema languages

One approach to face the compound document validation problem is to extend current schema languages to support multiple namespaces. Some schema languages even don't allow namespace prefixes in instances at all. For example the widely used DTD. Such languages cannot be directly used for compound document validation, but modern mainstream schema languages usually have namespace support included. XML Schema or Relax NG are cases in the point.

1.3.1.1. Relax NG

This section closely examines namespace capabilities of Relax NG; a popular full-featured nicely designed schema language (for more information see [RNG] or [HTML-VAL]). Relax NG is namespace aware which means elements and attributes are defined using their qualified names. The following example illustrates a straightforward way to define RDF inside the XHTML `head` section.

[3]Compound language is a term used within this text to describe a language composed of two and more different XML vocabularies. Such composition is considered to be a language itself, as it has its own syntax and semantics in addition to the syntax and semantics of the particular vocabularies.

Example 1.3. Allowing RDF inside the XHTML head

```
<define name="head.pattern">
  <element name="title" ns="http://www.w3.org/1999/xhtml"><text/></element>
  <interleave>
    ... more XHTML head definitions ...
   <element name="RDF" ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
     ... more RDF definitions ...
    </element>
  <interleave/>
</define>
```

The previous example is quite straightforward, but it is not really flexible indeed. To make it more flexible the schema needs to be modularized. Defining modules for each vocabulary allows to create schemas for various different compound languages on the fly simply by including the appropriate modules. Reusing the previous definition for a different combination of vocabularies would be very painful.

Modularizing the schema is no issue for Relax NG as it features a very nice modularity support. In Relax NG, "named patterns" are used for grouping definition fragments using the `define` element and assigning names to them. Two definition fragments with the same name are automatically combined. There are two strategies to combine definitions in Relax NG: `interleave` and `choice`. With `interleave`, the defined patterns can both occur in any order. `Choice` requires just one of the patterns to occur at a time. This mechanism allows to make parts of the definition more restrictive by including the right modules. Named patterns can also be made less restrictive by overriding them when included. The following example shows how to make the previous schema modular.

Example 1.4. Making the schema modular

Three abstract named patterns need to be defined first. One for all HTML block elements, second for all inline elements and third for the HTML `head` content. Each of those patterns represents a suitable context for placing different foreign vocabularies. `Interleave` combining method is specified for `head.pattern` as the head section is expected to have foreign elements in any order along with other HTML fragments. For `inline.pattern` and for `block.pattern` the `choice` method is used, thus the inline and block elements may contain either a foreign fragment or the eligible HTML elements.

```
...
<define name="head.pattern" combine="interleave">
  <zeroOrMore><ref name="head.foreign"/></zeroOrMore>
</define>
<define name="block.pattern" combine="choice">
  <ref name="block.foreign"/>
</define>
```

```
<define name="inline.pattern" combine="choice">
  <ref name="inline.foreign"/>
</define>
...
```

Further, each of the three patterns reference an additional named pattern which could be called a foreign pattern. Foreign patterns may be defined differently in different modules. The following module allows validation of HTML + RDF documents, where RDF may only occur in the context of the HTML `head` element. RDF is not only allowed there, it is also validated against the Relax NG schema for RDF (`rdfxml.rng`).

```
xhtml+rdf.rng
...
<define name="head.foreign" combine="interleave">
  <externalRef href="../foreign/rdf/rdfxml.rng" ▶
ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"/>
</define>
...
```

Different compound languages can be easily defined just by introducing a new module which adds some `externalRef` for the new language schema to the appropriate foreign patterns. For example the following foreign pattern definition allows SVG in any HTML block or inline element.

```
xhtml+svg.rng
...
<define name="block.foreign" combine="choice">
  <externalRef href="../foreign/svg/svg.rng" ▶
ns="http://www.w3.org/2000/svg"/ >
</define>
<define name="inline.foreign" combine="choice">
  <externalRef href="../foreign/svg/svg.rng" ▶
ns="http://www.w3.org/2000/svg"/ >
</define>
...
```

The next module allows MathML in any block or inline element.

```
xhtml+mathml.rng
...
<define name="block.foreign" combine="choice">
  <externalRef href="../foreign/mathml/mathml.rng" ▶
ns="http://www.w3.org/1998/Math/MathML"/ >
</define>
<define name="inline.foreign" combine="choice">
```

```
    <externalRef href="../foreign/mathml/mathml.rng" ▶
  ns="http://www.w3.org/1998/Math/MathML"/ >
  </define>
  ...
```

When including all of the three introduced modules into the modular schema, the result is a compound language schema based on XHTML which allows RDF meta-data in the header and SVG or MathML fragments in the document's body. For any new combination of languages, just a simple Relax NG schema with all the appropriate includes needs to be created. Below is an example of such schema for the XHTML + RDF + SVG + MathML compound language.

```
xhtml+rdf+svg+mathml.rng
...
<include href="../foreign/xhtml+rdf.rng"/>
<include href="../foreign/xhtml+svg.rng"/>
<include href="../foreign/xhtml+mathml.rng"/>
...
```

There is another Relax NG feature especially useful for compound documents. Name classes allow to use wild-cards for element and attribute qualified names. This means, vocabularies are validated laxly. The `anyName` wild-card matches any qualified name and the `nsName` wild-card matches qualified names within the specified namespace. In the next example, arbitrary foreign vocabularies are allowed in the context of the HTML `head` element.

Example 1.5. Arbitrary vocabularies

The following pattern allows to combine XHTML elements in the `head` context with arbitrary vocabularies. All elements and attributes are matched, except those assigned into the HTML namespace, because they should not be validated laxly. The `foreign.head` pattern is referenced recursively to allow an arbitrary structure of foreign elements.

```
    ...
  <define name="head.foreign" combine="interleave">
    <element>
      <anyName> <!-- any element is allowed expect HTML elements -->
        <except>
          <nsName ns="http://www.w3.org/1999/xhtml"/>
        </except>
      </anyName>
      <zeroOrMore>
        <choice>
          <attribute>
            <anyName>
```

```
            <!-- any attribute is allowed expect HTML attributes -->
              <except>
                <nsName ns="http://www.w3.org/1999/xhtml"/>
              </except>
            </anyName>
          </attribute>
          <text/>
          <!-- recursive reference zero or more any element children -->
          <ref name="head.foreign"/>
        </choice>
      </zeroOrMore>
    </element>
  </define>
```

Similar features for defining compound languages demonstrated previously using Relax NG can also be found in the widely used and supported XML Schema.

1.3.2. Different approach

The previous section has shown compound document schemas created using namespace support in Relax NG. This approach is straightforward and convenient, especially for people who are already familiar with this popular schema language. But there are several disadvantages.

To allow a foreign vocabulary in some contexts of XHTML (as demonstrated in example Example 1.4, “Making the schema modular”) is a simple task for someone who is familiar with the implementation details of the XHTML schema (knowing the structure of the schema's definition patterns and modules), but it is less straightforward for someone who doesn't have the right insight.

Moreover, using the approach shown in Section 1.3.1, “Current schema languages” doesn't allow to reuse existing schemas for languages to be combined. Standalone language schemas aren't often prepared to be combined with other schemas[4]. In addition, they are frequently written in different schema languages or even in languages which aren't namespace-aware at all, thus they may not be combined without conversion[5]. All those factors cause reusing of such schemas to be very inconvenient and time-consuming in some cases. It may require redesigning parts of the schemas and converting all schemas to one common schema language which features namespace support.

Such approach would not only require deep knowledge of the particular schemas and a long implementation time, but it also leads to maintenance issues. As different

[4]Standalone language schemas don't usually have the right level of modularity and abstraction which is needed for seamless integration which other schemas.
[5]When combining schemas using a namespace-aware language, all external referenced schemas must be written in the same schema language.

languages evolve over time, constant updating of the redesigned or converted schemas is required. This issue is especially pressing for complex compound languages.

To conclude, reusability of existing schemas is an important requirement which is not satisfied within today's namespace-aware schema languages (e. g. Relax NG or XML Schema). To allow schema reusability, a completely different approach to compound document validation is needed; an approach which is completely independent on the schema languages used to define the particular vocabularies and on their schema implementation. Those objectives can not be simply met without a special single-purpose "meta-schema" language intended to express eligible ways to combine different vocabularies in one document.

Such language would bring significant benefits over the single namespace-aware language approach. It allows people designing XML vocabularies to fully focus on the schema, without even thinking about how can their vocabulary possibly be combined with a different one. They don't need to design any additional modules or definitions and they may select a schema language which best suits their needs. It doesn't even need to be a mainstream language, nor a namespace-aware language.

People designing compound languages would benefit from such approach as well. They may fully focus on declaring the particular rules for combining different vocabularies without having any additional knowledge about their schema implementation details. They don't even need to understand the different schema languages at all.

Separating vocabulary constraints from compound language constraints also simplifies schema maintenance. Changes in any particular vocabulary schema doesn't imply any changes in the compound language definition. Introducing a new schema for an arbitrary compound language doesn't require any adjustments in the vocabulary schemas. Basically all schemas used to describe a particular compound language and its vocabularies are completely independent which allows them to be reused as they are.

1.3.3. Evolution of the alternative

The alternative concept of compound document validation described in the previous section originated in 2001 when Murato Makoto introduced the Relax Namespace (see [RNS]). The purpose was to bring namespace support to the Relax schema language (a predecessor of Relax NG). The Relax Namespace idea is based on the "divide and validate" concept. Compound documents are first decomposed into "islands", where each "island" is a single-namespace well-formed XML fragment. Such fragments may be further validated against the appropriate single-namespace schemas.

Over the following years, other well-known XML experts, mainly James Clark and also Rick Jelliffe, enhanced the original concept. As a result the Modular Namespaces [MNS], Namespace Switchboard [NSSB] and Namespace Routing Language [NRL] have been introduced. Modular Namespaces brought more control over the "divide and validate" process using a simple language. In the simplest form, such language

maps namespace URIs to schema URIs to express what schema shall be used to validate different vocabularies. This mapping is not fixed, it may differ in different contexts within the validated document. Modular Namespaces also introduced the concept of schema language transparence. The importance of such concept for schema reusability has been discussed in the previous section.

The concept did further evolve with Namespace Switchboard which made the language simpler and clearer by moving the expressive power from the low-level validation details more to the user perspective. Namespace Switchboard as well as Modular Namespaces both served as a base for the Namespace Routing Language (NRL). NRL is a mature feature-rich language. It takes advantages from both it's predecessors. Simplicity is mostly inherited from Namespace Switchboard and features from Modular Namespaces.

Recently, the NRL concept has been further slightly fine-tuned by dropping some features and introducing new ones. The result has been internationally standardized as a compound document validation approach which is called the Namespace-based Validation Dispatching Language (NVDL). The following chapter explains the NVDL specification in detail.

Chapter 2

NVDL

NVDL, which means Namespace-based Validation Dispatching Language, is "Part 4 of ISO/IEC 19757 DSDL" (Document Schema Definition Languages) international standard. NVDL is a simple "meta-schema" language which allows to control processing and validation of compound documents. Figure 2.1, "NVDL validation process at a glance" demonstrates a particular validation dispatching process decomposed into several phases.

The essence of NVDL is dividing XML compound document instances[1] into sections[2] each of which contains elements or attributes from a single namespace. A section tree is first constructed for every instance. Sections are further combined or manipulated in various ways to create so called validation candidates.

Manipulation of sections is achieved through rules and their corresponding actions defined in an NVDL script. Actions are executed on a particular section whenever it matches a certain rule; usually in case the sections namespace matches the rule's namespace wildcard.

There are several actions defined in NVDL; e. g. `attach` for attaching sections back to their parent, `unwrap` to handle wrapped sections and `validate` to send a particular validation fragment to a particular validator.

After executing actions, validation candidates are created (those are usually single namespace fragments) and they are further filtered for redundancy into validation fragments. Such fragments are finally independently send for validation against different subschemas[3]. The NVDL validation process is examined in detail in Section 2.2, "Specification", but a complete specification is contained within [NVDL].

Figure 2.1, "NVDL validation process at a glance" shows a compound document with two vocabularies. In the first phase, the document is decomposed into sections depending on the namespace of the different fragments. According to the NVDL script,

[1] The term instance is used for the input document of the validation process

[2] In NVDL, there are element sections and attribute sections. An element section is defined as an element such that a single namespace applies to itself and to all its descendant elements. An attribute section is basically a non-empty set of attributes having the same namespace.

[3] Subschema is defined in the NVDL specification as a schema referenced by the NVDL script.

different actions (`validate`, `unwrap` and `attach`) are executed on the particular sections. This leads to construction of validation candidates. Finally, for every `validate` action a non-redundant candidate is send for validation. Appendix A, *NVDL validation dispatching process* shows a corresponding XML instance and NVDL schema example.

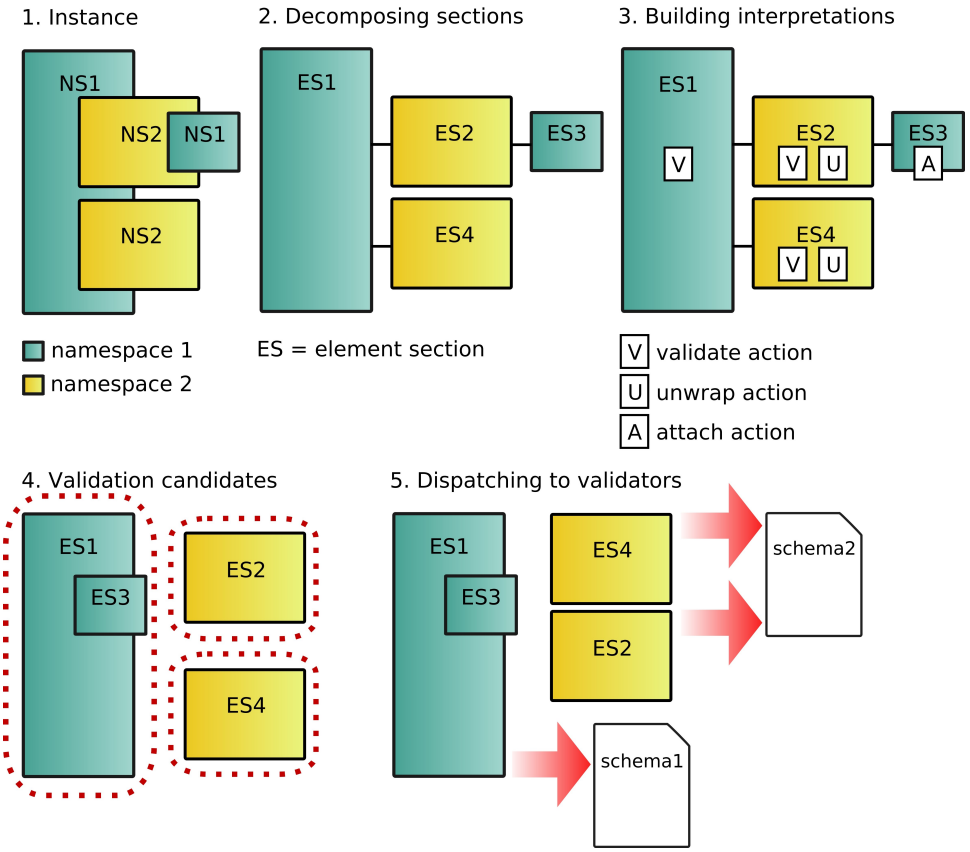

Figure 2.1. NVDL validation process at a glance

When compared to other approaches, NVDL offers many advantages. It features a standardized and easy to understand language to define compound document validation dispatching processes. With NVDL, different vocabularies may be easily allowed, banned or send for further validation depending on the particular context where they occur within the validated instance. NVDL language semantics is described in Section 2.1, “Semantics in example”.

The ability to create single namespace fragments allows not to care about namespaces in the subschemas at all. Single namespace schemas are easier to write and what is important also easy to reuse for various different compound languages. Moreover, NVDL is not bound to a specific schema language, different schema languages can be used in combination during a single validation process. Subschemas may be written in any preferable schema language e. g. Relax NG, XML Schema, Schematron or DTD. This is again important in terms of reusability, because in the real world, XML vocabularies are usually described using different schema languages. NVDL allows to reuse those schemas as they are. There is no need for converting or modifying them.

When having a set of subschemas for different vocabularies, using NVDL, it is straightforward to create various NVDL definitions for all different combinations of such vocabularies without the need to introduce any changes to the particular subschemas at all.

Using entirely Relax NG or XML Schema for compound document validation leads to uniformity as it forces users to convert schemas for different vocabularies to the same language. NVDL, on the other hand, means variety, as it allows to choose the schema language with best suits the particular vocabulary needs. There is absolutely no need to choose a mainstream language.

2.1. Semantics in example

This section describes the semantics of NVDL. Different elements and attributes and their meanings are explained with the help of example NVDL schemas. Simple scenarios are used first and gradually more difficult and specific ones are introduced. A similar concept is used in [NRL] which served as a base for this section. The first example is probably the simples NVDL schema imaginable.

Example 2.1. The "Hello World!" NVDL schema

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
<namespace ns="http://hello-world">
  <validate schema="hello-world.dtd"/>
</namespace>
</rules>
```

The root element in any NVDL schema is called `rules`. It contains all the rules that determine the validation process execution. In Example 2.1, "The "Hello World!" NVDL schema", there is just one rule. Whenever one of the validated XML instance sections matches the namespace `http://hello-world`, the whole section is send for validation against the `hello-world.dtd` subschema. Elements from different namespaces are rejected, which is the default behavior.

This example is basically equivalent to classical single-namespace validation. The validation process is the same, as if validating the instance directly against the `hello-world.dtd` schema using a DTD validator. Let's move to a more realistic example expecting more namespaces in a single instance; basically doing a simple compound document validation.

Example 2.2. Compound document schema with multiple `namespace` rules

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.rng"/>
```

```
    </namespace>
    <namespace ns="http://www.w3.org/2000/svg"/">
      <validate schema="svg.sch"/>
    </namespace>
  </rules>
```

In Example 2.2, “Compound document schema with multiple `namespace` rules”, there are two different `namespace` rules each of which contains a `validate` action. This example shows a basic mapping between a namespace URI (specified by the `ns` attribute) and a schema URI (specified by the `schema` attribute).

The meaning of the NVDL script is very simple. Each section which belongs to the XHTML namespace is validated against the `xhtml.rng` Relax NG schema and each SVG section against the Schematron rules defined in `svg.sch`.

2.1.1. Adjusting validation

Validation of sections against subschemas can be adjusted using several attributes or children of the `validate` element. Not only the `schema` attribute can be used to specify subschemas. In some cases, it is useful to use the `schema` element instead, to embed a subschema directly into the NVDL script. The `schema` element may contain either text, in case the subschema is not XML-based[4] or a foreign XML fragment. When embedding subschemas directly into NVDL, such NVDL schema becomes effectively a compound document. This implies, an NVDL schema for NVDL can be defined as demonstrated in Appendix C, *NVDL schema as a compound document*.

One of the problems an NVDL dispatcher[5] has to solve is, what validator to invoke for each subschema. In most cases subschemas are defined in the XML format, thus the schema language can be easily recognized from the subschema's parent element namespace. But in some cases, the subschema is in a different format and the NVDL dispatcher has to determine the schema language from the MIME type. In case the MIME type in not available, the schema language should be manually specified in the NVDL script using the `schemaType` attribute.

A typical example of a non-XML schema language format is DTD. In this case, the value of the `schemaType` attribute is `application/xml-dtd`. For Relax NG in the compact syntax the value is `application/x-rnc`.

Some validators use specific options to adjust the validation process. Such options can be specified directly in an NVDL script. The NVDL dispatcher takes care of passing those into the appropriate validator. Options are expressed using `option` elements inside the `validate` action. Their name and value pairs are set using the `name` and `arg` attributes. If the validation process requires the validator to support a particular option, the

[4] This applies e. g. to DTD or Relax NG in the compact syntax.
[5] An NVDL dispatcher is an application doing validation dispatching in compliance with the NVDL specification.

`mustSupport` attribute should be set to `true`. An error is returned, if the validator doesn't support it.

Frequently, it is needed to allow or reject some sections in a particular context without validating them. It doesn't make much sense to express such behavior using some specific schema language. For this purpose NVDL offers predefined schemas. Instead of the `validate` action, `allow` or `reject` can be used directly. In example Example 2.3, "Predefined schemas", XHTML sections are validated using the `xhtml.rng` schema, but all SVG sections are allowed without even attempting to validate them. All other element sections are rejected using the `anyNamespace` rule in combination with `reject`. Such definition is in fact redundant, as it is NVDL's default behavior to reject any section which doesn't match any of the defined rules.

Example 2.3. Predefined schemas

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.rng"/>
  </namespace>
  <namespace ns="http://www.w3.org/2000/svg">
    <allow/>
  </namespace>
  <anyNamespace>
    <reject/>
  </anyNamespace>
</rules>
```

There is another interesting NVDL feature related to validation; for one namespace, several `validate` actions can be defined. This tells the NVDL dispatcher to invoke several validators for every matching section. As different schema languages are more or less suitable to express some kind of constraints, this is a reasonable use-case. Sometimes better validation results are achievable through a combination of two or more schema languages[6].

One of the promising schema language combinations is for example Relax NG and Schematron. Where Relax NG is suitable to define the elementary grammar of a vocabulary (mostly parent-child relations), Schematron is especially useful to express complex validation rules across the XML instance tree.

In some cases it makes a good sense to ensure not only validity of HTML documents but also accessibility, which can be defined as compliance with the Web Content Ac-

[6]Not every schema language combination is automatically useful. Some languages are more suitable to be combined than others. In general, there are two major categories of schema languages: grammar-oriented and rule-based. Usually it makes more sense to combine languages from different categories. More information on this topic are available in [HTML-VAL].

cessibility Guidelines[7] (WCAG). Where Relax NG is the right solution to express XHTML grammar, Schematron is the preferable language for expressing the various complex accessibility rules. As shown in the next example, multiple `validate` elements cause XHTML sections to be validated against both schemas.

Example 2.4. Multiple `validate` elements for a single namespace

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.rng"/>
    <validate schema="wcag.sch"/>
  </namespace>
</rules>
```

2.1.2. Rules

Rules are represented either by the `namespace` or `anyNamespace` element and they consist of a condition and a list of actions. The rule is triggered whenever an element or attribute section matches the given condition. In this case, each action in the list is executed. There are several different types of actions in NVDL. One of them is for example the `validate` action which was mentioned earlier.

Rule conditions are defined using a namespace URI. The `namespace` rule is applicable to any section whose namespace matches the value of the rule's `ns` attribute. The `anyNamespace` rule works differently. As it matches every section which doesn't have any applicable `namespace` rule defined, it basically specifies default behavior for vocabularies which are not explicitly listed. Example 2.3, "Predefined schemas" illustrated the use of `anyNamespace` rule in conjunction with `reject`. When `allow` is used instead, the default strict validation behavior is overridden and NVDL validates laxly. This means, any section in an arbitrary namespace with no matching rule is automatically allowed without being validated.

Sometimes, it is desirable to match several namespaces whose URI matches a special pattern. For that reason, NVDL introduces wild-cards. The `wildCard` attribute at `namespace` rules sets a special symbol (one character) that stands for one or more unspecified characters in the namespace URI. If `wildCard` is not present, the default wild-card symbol is a star `*`. Wild-cards are useful for example in cases, when the same schema (or behavior) applies to several versions or mutations of a vocabulary and their namespace URI slightly differs.

Rules can match elements, attributes or both. Implicitly they match elements. This means, rules apply by default to element sections only. But this can be altered using

[7] Web Content Accessibility Guidelines is a set of recommendations aimed to make Web content accessible to people with all sorts of disabilities. For more information refer to [WCAG].

the `match` attribute at both, `namespace` or `anyNamespace` rules. The `match` attribute accepts "element", "attribute" or "element attribute" values.

2.1.3. Modes

In the previous examples, just simple global rules applicable for all element and attribute sections in the entire document were considered. Modes bring much more flexibility and a fine-grained control over the validation process. Different rules can be grouped within modes and such groups are applicable in different context of the document.

The previous examples, where `namespace` and `anyNamespace` elements were contained directly within the root `rules` element, may be understood as NVDL scripts with just one global mode applicable in any context. When using multiple modes, the root element doesn't contain different rules directly. Instead, it contains several `mode` children which than contain the actual rules. In this scenario a `startMode` attribute has to be present at the `rules` element to specify the initial mode.

Every time an action is executed on a particular section, the NVDL dispatcher transits from one mode to another. There are basically two possibilities for specifying transitions between modes. Every action (e. g. the `validate` action) can have a `useMode` attribute, which references a different mode using its unique name[8]. Another approach is nesting modes directly into actions. There is one important difference between those two approaches. Where named modes can be referenced by multiple actions, transition to a nested mode is only possible by executing its parent action. If no `useMode` or any nested `mode` is defined for an action, the action transits by default back to the same mode.

Example 2.5, “Using modes” examines a simple NVDL script which ensures, instances have SVG sections nested into XHTML, not the other way around. Only XHTML namespace sections are allowed in the initial mode. For any nested section, the validation process transits to the `nested` mode which allows only the SVG namespace. In fact, SVG fragments are prevented from containing any further foreign namespace fragments.

Example 2.5. Using modes

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="init">
  <mode name="init">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <validate schema="xhtml.rng" useMode="nested"/>
    </namespace>
  </mode>
  <mode name="nested">
```

[8]Names are assigned to modes using their `name` attribute. Named modes are always children of the `rules` element.

```
      <namespace ns="http://www.w3.org/2000/svg"/">
        <validate schema="svg.sch"/>
      </namespace>
    </mode>
  </rules>
```

Readability of this example can be enhanced using nested modes. Nesting modes does not change the meaning of the script, it is just a different way to describe the same behavior.

Example 2.6. Using nested modes

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="init">
  <mode name="init">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <validate schema="xhtml.rng">
        <mode>
          <namespace ns="http://www.w3.org/2000/svg"/">
            <validate schema="svg.sch"/>
          </namespace>
        </mode>
      </validate>
    </namespace>
  </mode>
</rules>
```

As shown in Example 2.6, "Using nested modes", the `mode` element can appear as a child of the `rules` element or it can be nested inside actions.

In addition, there is one more place where modes can appear; included inside other modes. In such case, the NVDL dispatcher takes care of merging those modes together into a single mode. Conflicts are resolved. Every child mode rule which defines the same condition as one of the parent mode rules is overridden.

Example 2.7. Mode inclusion

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="init">
  <mode name="init">
    <mode>
      <anyNamespace>
        <allow/>
      </anyNamespace>
    </mode>
```

```
      <anyNamespace>
        <reject/>
      </anynamespace>
    </mode>
  </rules>
```

The previous NVDL script is equivalent to the following one. Both scripts simply reject every instance.

Example 2.8. Merged mode

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="init">
  <mode name="init">
    <anyNamespace>
      <reject/>
    </anynamespace>
  </mode>
</rules>
```

Mode inclusion is especially useful when including external modes using XInclude. With XInclude, only well-formed XML fragments can be included. This means, different rules cannot be included directly, but instead they need to have a root element. It is straightforward to encapsulated rules intended for inclusion into a parent `mode` element.

Mode inclusion is a nice way to achieve mode inheritance[9]. The Example 2.6, "Using nested modes" can be further extended to allow both XHTML as well as SVG fragments to have nested RDF meta data sections. The next example shows the solution.

Example 2.9. Mode inheritance

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
xmlns:xi="http://www.w3.org/2001/XInclude" startMode="init">
  <mode name="init">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <validate schema="xhtml.rng">
        <mode>
          <xi:include href="rdfmode.xml"/>
          <namespace ns="http://www.w3.org/2000/svg"/">
            <validate schema="svg.sch">
              <mode>
                <xi:include href="rdfmode.xml"/>
              </mode>
```

[9] In NRL, mode inheritance is directly supported as part of the language. The `extends` attribute is used to reference parent modes. For more information refer to [NRL]

```
            </validate>
          </namespace>
        </mode>
      </validate>
    </namespace>
  </mode>
</rules>

rdfmode.xml
<mode>
  <namespace ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <validate schema="rdf.rng"/>
  </namespace>
</mode>
```

2.1.4. Attaching sections with `attach`

The previous text mentioned actions in general, but only the `validate` action was introduced and explained in detail. Several other different actions and their meanings are explained in the following text. One of them is the `attach` action which allows re-attaching child sections[10] back to their parent to make them possibly validated together as one fragment.

When attaching sections, XML fragments with multiple namespaces are created. This may at first look like going against the principal of NVDL, as NVDL is all about separating different namespace fragments. But as already discussed, it is not just NVDL which can handle compound document validation. Modern validation languages, such as Relax NG or XML Schema, can cope very well with compound documents. In case there is already a nicely designed compound document schema, it does make a good sense to use it.

In case there is a well designed schema for XHTML in Relax NG, which defines abstract classes for inline and block elements as seen in Example 1.4, "Making the schema modular", it is easy to allow nested SVG fragments to occur just in the context of an inline or block element, without the need of listing all such elements explicitly. This would be necessary in case pure NVDL is used to describe the context in which SVG can occur. In this case, it makes sense to use the Relax NG compound document schema and use the `attach` action inside the SVG namespace rule as shown in Example 2.10, "Attaching SVG sections back to XHTML".

[10] In this text, a child element section for section `s` is understood as every element section which is referenced by section `s` in the decomposed section tree.

Example 2.10. Attaching SVG sections back to XHTML

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="xhtml">
<mode name="xhtml">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml+svg.rng"
              useMode="svg"/>
  </namespace>
</mode>
<mode name="svg">
  <namespace ns="http://www.w3.org/2000/svg">
    <attach/>
  </namespace>
</mode>
</rules>
```

2.1.5. Unwrapping element sections

When combining different XML vocabularies, foreign elements or attributes are usually embedded as children into different contexts of the parent language. A more difficult scenario occurs in case elements of one language are wrapped around fragments of the language to be validated. Such problem occurs for example in case when using XML-based scripting or templating languages. For example XHTML fragments in an XSLT stylesheet are wrapped by the XSLT vocabulary.

This is exactly the right situation for the NVDL `unwrap` action. With `unwrap`, the current section is left out completely from a potential validation candidate. This is in particular useful in combination with `attach`. Invoking `unwrap` on the current section and `attach` on one of its child sections means, the child section is attached directly to the parent of the current section. This approach allows templating language sections to be filtered out completely leaving just the wrapped content for validation. Templating languages are often used for XHTML-based styling of domain specific XML languages using XSTL stylesheets. This is also the case in Example 2.11, “Validating XHTML wrapped by XSLT”.

Example 2.11. Validating XHTML wrapped by XSLT

This NVDL schema simply filters every occurrence of XSLT out. The pure XHTML is than send for validation against the `xhtml.rng` schema.

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="root">
  <mode name="root">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <validate schema="xhtml.rng"
```

```
                useMode="xslt"/>
    </namespace>
    <namespace ns="http://www.w3.org/1999/XSL/Transform">
      <allow/>
    </namespace>
  </mode>
  <mode name="xslt">
    <namespace ns="http://www.w3.org/1999/XSL/Transform">
      <unwrap/>
    </namespace>
    <namespace ns="http://www.w3.org/1999/xhtml">
      <attach/>
    </namespace>
  </mode>
</rules>
```

The `unwrap` concept works nicely in simple situations, but in complex scenarios difficulties are likely to emerge. With templating languages it is hard to predict the execution flow. Problems can arise even in case of a simple if-else condition which is a must-have feature of any templating language. When if-else mark-up is unwrapped, both normally disjoint conditional clauses are attached into the validated fragment. This can cause problems when using schema languages to control element's number of occurrence; especially when just one occurrence is allowed.

Example 2.12. XHTML wrapped by XSLT

The following XSLT stylesheet produces XHTML documents with either the "Book" title, the "Magazine" title or simply an "Item" title, depending on the root element of the input document. Just one condition is met at a time, thus the resulting XHTML document always has just one `title` element. A different situation occurs when unwrapping the condition mark-up as shown in the following example.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" ▶
xmlns="http://www.w3.org/1999/xhtml" version="1.0">
  <xsl:template match="/">
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
       <xsl:choose>
         <xsl:when test="book">
           <title>Book</title>
         </xsl:when>
        <xsl:when test="magazine">
          <title>Magazine</title>
        </xsl:when>
        <xsl:otherwise>
          <title>Item</title>
```

```
          </xsl:otherwise>
        </xsl:choose>
        </head>
        <body>
            <p>
              ..
            </p>
        </body>
      </html>
    </xsl:template>
  </xsl:stylesheet>
```

Here is the resulting XHTML fragment after unwrapping XSLT.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Book</title>
    <title>Magazine</title>
    <title>Item</title>
  </head>
  <body>
      <p>
        ..
      </p>
  </body>
</html>
```

Even the XSLT stylesheet always produces a valid XHTML, after `unwrap` is used, an invalid validation fragment is produced. According to the XHTML specification, just one `title` element is allowed inside `head`.

As a workaround, a modified version of the XHTML schema which allows any number of elements in any context could be used. In simple scenarios, unwrapping can be also fine-tuned using the NVDL `context` element which is discussed later.

When validating wrapped content, several other issues not mentioned in this section can occur. `Unwrap` may not be powerful enough to solve all such issues and in most cases it may be simply preferable to validate the wrapped content simply after being processed by the templating engine. Still there can be other reasonable use-cases for `unwrap` in the real world.

2.1.6. The `attachPlaceholder` action

Where the `attach` action attaches the whole section to its parent, `attachPlaceholder` attaches just a special `placeholder` element. The `placeholder` element is assigned to

the `http://purl.oclc.org/dsdl/nvdl/ns/instance/1.0` namespace and it has two attributes. The `ns` attribute specifies the section's namespace URI and the `localName` attribute contains the local name of the section's root element.

The `placeholder` element can than be defined in the subschema to check the context of different foreign fragments without actually validating them within this particular subschema.

The three recently discussed actions: `attach`, `unwrap` and `attachPlaceholder` are so called “no result actions” as they don't directly result in validation of anything. For obvious reasons, those actions are mutually exclusive and thus just one of them can be present in the same rule.

2.1.7. Canceling nested actions

The `cancelNestedAction` element may occur inside rules on the same place where usually actions occur, but it is not an action itself. It basically prevents any action to be executed. When `cancelNestedAction` is present, neither other `cancelNestedAction` element, nor any actions may occur in that particular rule.

`cancelNestedAction` is in particular useful when having a general rule in place, but an exception for some namespace needs to be defined. The following example illustrates this situation. Any namespace is attached to the parent section, but the SVG sections are not, because they shall not be validate with the same subschema.

Example 2.13. Do not attach SVG fragments to XHTML

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="root">
<mode name="root">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.rng"
              useMode="attach"/>
  </namespace>
</mode>
<mode name="attach">
  <namespace ns="http://www.w3.org/2000/svg">
    <cancelNestedAction/>
  </namespace>
  <anyNamespace>
    <attach/>
  </namespace>
</mode>
</rules>
```

There is one more note to make about `cancelNestedAction`. If mode inclusion is used, `cancelNestedAction` rules are left out from any included child modes.

2.1.8. Context dependent processing

Modes offer a mechanism to change the NVDL dispatcher behavior during the validation process depending on the context of the currently processed section. In previous examples, the initial mode for child sections was always determined using the action's default transition defined by `useMode`. This is also an example of context specific processing, where the context is specified as the namespace of the parent section. When looking back at Example 2.10, "Attaching SVG sections back to XHTML", this NVDL schema may be interpreted as follows: Do attach SVG sections back to their parent whenever they are in the *context* of an XHTML section and reject every SVG sections in any other *context*.

Expressing current section's context as the parent section namespace is likely to be sufficient in most cases, but sometimes it may be convenient to specify context more precisely, for example as an element path within the parent section.

In NVDL, any action can have several `context` child elements with a `path` attribute and a mode transition. The transition is defined analogically to default transitions at actions. An `useMode` attribute or a nested `mode` element can be used.

The `path` attribute defines the context path within an element section[11]. The syntax is inspired by XPath, but it is much simpler. As context is specified within one single-namespace element section, there is no need to use any namespace prefixes. Moreover, there are no axis, functions and other advanced XPath constructs. The `path` attribute value is basically a list of one or more choices separated by the "`|`" delimiter. Each choice is than a list of local element names splitted with a path separator "/". If preceded by a path separator the choice is considered to be an absolute path (a path from the root element of the parent section) otherwise the path is relative.

Every time an action is executed for an element section, the NVDL dispatcher first goes through the list of its `context` children. If any of the `path` expressions matches the current section's context path, the transition at this particular `context` is invoked. Otherwise the action's default transition is used. Section's context path consists of all element local names in the branch which begins at the parent section root element and ends at the former parent element of the section.

With `context`, vocabularies can be handled differently when embedded in different places (nodes) within the parent vocabulary section. Precise context handling is convenient for example when validating XHTML documents with embedded RDF. As meta-data are not intended to be rendered, they shall not occur in the document's `body`.

[11] A path for element section `s` is constructed as a sequence of element local names. Such sequence begins with the `s`'s parent element section's root element and it ends with the local name of the former (before decomposition) parent element of the section `s`.

The right place to put meta-data is apparently in the `head` section. The following example shows such constraints expressed using NVDL.

Example 2.14. Allow RDF in the `head` context only

RDF fragments are allowed just inside the `head` element. In this case, RDF is allowed inside any `head` element across the document. This is sufficient, as occurrence of `head` in a wrong place would not pass validation against the `xhtml.rng` schema. A similar result can be achieved using absolute path `/html/head` to define the context. In this situation RDF fragments can just occur in that particular `head` node.

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="root">
  <mode name="root">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <validate schema="xhtml.rng">
        <context path="head"
                 useMode="rdf"/>
      </validate>
    </namespace>
  </mode>
  <mode name="rdf">
    <namespace ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      <validate schema="rdf.rng" useMode="attach"/>
    </namespace>
  </mode>
  <mode name="attach">
    <anyNamespace>
      <attach/>
    </anyNamespace>
  </mode>
</rules>
```

2.1.9. Working with attributes

In the previous text, no difference was made between attribute and element sections. By default, NVDL attaches attribute sections back to their element sections, but using `match` attribute at rules, attribute sections may be handled specifically. Each attribute section consists of attributes belonging to the same namespace and having the same parent element. When using `match="attribute"`, `validate` action may be invoked directly on an detached attribute section. This basically means, a standalone attribute set is send to be validated against the specified subschema.

A standalone set of attributes is not considered to be a well-formed XML. That's the reason why NVDL creates a meaningless element which is called the `virtualElement` based in the `http://purl.oclc.org/dsdl/nvdl/ns/instance/1.0` namespace. Validated

attributes from the attribute section are than attached to the `virtualElement` before being sent for validation.

Modern validation languages e. g. Relax NG or XML Schema have a very expressive constructs to constraint attribute values using data types. Specific handling of attribute sections allows to define such constrains in a separated subschema. A nice example are attributes from the XML default namespace `http://www.w3.org/XML/1998/namespace` (described in [NS]). It is undoubtedly convenient to have a specific subschema for those attributes, as they are expected to occur in most compound languages. With NVDL, such subschema can simply be reused every time those attribute values are required to be validated. There is no need to have them defined again and again in every specific subschema.

NVDL expects attribute-only subschemas to have no supplemental elements defined. To make validation using such subschema possible, NVDL first performs a schema language specific transformation which allows attributes to be attached to the `virtualElement`. NVDL specification contains as an example the Relax NG specific transformation. The schema is wrapped by `<element><anyName/>...content of the original schema...</element>`. Using `anyName` allows any element to contain the defined attributes. For different schema languages, NVDL implementations should introduce different transformation rules.

Example 2.15. Validation of default XML attributes

The NVDL schema in this example validates the values of the default XML attributes. The attributes aren't attached back to their elements, thus they don't need to be defined in the `some-language-schema.rng` schema.

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="... some language namespace ...">
    <validate schema="some-language-schema.rng"/>
  </namespace>
  <namespace ns="http://www.w3.org/XML/1998/namespace" match="attributes">
    <validate schema="xmlattr.rng"/>
  </namespace>
</rules>
```

Below is a fragment of the `xmlattr.rng` subschema. For example the `xml:space` attribute value is constrained to contain either `preserve` or `default`, the `xml:lang` attribute shall contain only valid language codes and the `xml:base` attribute a valid URI.

```
<group datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
     xmlns="http://relaxng.org/ns/structure/1.0">
  <optional>
    ...
    <attribute name="xml:space">
```

```
        <choice>
          <value>preserve</value>
          <value>default</value>
        </choice>
      </attribute>
      <optional>
        <attribute name="xml:lang">
          <choice>
            <data type="language"/>
            <value/>
          </choice>
        </attribute>
      </optional>
      <optional>
        <attribute name="xml:base">
          <data type="anyURI"/>
        </attribute>
      </optional>
      ...
    </optional>
  </group>
```

Before validation, the NVDL dispatcher transforms `xmlattr.rng` subschema into the following form.

```
  <element datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
       xmlns="http://relaxng.org/ns/structure/1.0">
    <anyName/>
    <group>
      <optional>
        ...
        <attribute name="xml:space">
          <choice>
            <value>preserve</value>
            <value>default</value>
          </choice>
        </attribute>
      <optional>
        <attribute name="xml:lang">
          <choice>
            <data type="language"/>
            <value/>
          </choice>
        </attribute>
      </optional>
      <optional>
```

```
        <attribute name="xml:base">
          <data type="anyURI"/>
        </attribute>
      </optional>
        ...
      </optional>
    </group>
  </element>
```

2.1.10. When lacking namespaces

In some cases, it is not possible to rely solely on namespaces to decompose documents into sections. Some languages don't use namespaces at all (for example DocBook 4.2). In this case, some other mechanism to extract sections is needed. NVDL offers the `trigger` construct to achieve that.

`Trigger` elements can occur as children of the `rules` element. There are two obligatory attributes on `trigger`. The `ns` attribute specifies the namespace and the `nameList` attribute is a space separated list of element local names. A trigger is fired for any element whose namespace exactly matches the one specified and whose local name is contained in the `nameList`. Also the element shall not be a root of the current element section and its parent shall not be located by the same `trigger`. An example validation dispatching processes using triggers is shown in Appendix D, *Validation using triggers*.

2.1.11. Annotating NVDL

There are several different possibilities to annotate an NVDL schema. For example, every action (`attach`, `attachPlaceholder`, `unwrap`, `allow`, `reject` or `validate`) can have a `message` element or attribute associated with it. Comments and hints related to the particular action may be placed here, but it is important to keep in mind such messages may be displayed by the NVDL dispatcher during the validation dispatching process whenever a particular action gets executed.

Further, any annotation language in a different namespace than the NVDL namespace can be used to put additional annotations within nearly any context of the NVDL script. For example the annotation language based in `http://relaxng.org/ns/compatibility/annotations/1.0` designed to be used in Relax NG can make NVDL schemas easily understandable and more human-readable. Beside annotations, NVDL scripts may also be commented using traditional XML comments.

Example 2.16. An annotated and commented NVDL script

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0" ▶
startMode="root">
```

```
  <!-- NVDL schema for XHTML with embedded RDF meta-data allowed within ▶
the XHTML header -->
  <mode name="root">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <a:documentation>XHTML is the parent language, no other language ▶
is allowed in this context.</a:documentation>
     <validate schema="xhtml.rng" message="Sending an XHTML fragment for ▶
validation.">
        <context path="head" useMode="rdf">
          <a:documentation>RDF may occur only in the head ▶
context.</a:documentation>
        </context>
      </validate>
    </namespace>
  </mode>
  <mode name="rdf">
    <namespace ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
     <a:documentation>Only RDF is allowed within XHTML.</a:documentation>
     <validate schema="rdf.rng" useMode="attach" message="Sending an RDF ▶
fragment for validation."/>
    </namespace>
  </mode>
  <mode name="attach">
    <anyNamespace>
      <a:documentation>Any foreign vocabulary is allowed within ▶
RDF.</a:documentation>
     <attach message="Attaching foreign vocabularies within RDF back to ▶
validate them together."/>
    </anyNamespace>
  </mode>
</rules>
```

2.2. Specification

This section is basically an introduction to the NVDL specification in a nutshell. Detailed information are contained within [NVDL]. The NVDL specification does not describe the language semantics directly, it is rather a formal description of the NVDL validation and dispatching process and the NVDL language syntax. The specification is very low-level, it defines a formal framework of the process to ensure consistent behavior of different NVDL implementations. It does not mention any NVDL use-cases nor its high-level features.

The validation dispatching process is divided into several tasks. First, a data model representation is built from XML infosets for the NVDL schema and the validated instance. The schema is then transformed into simple syntax for easier processing. In the next phase, the instance is decomposed into element and attribute sections. The NVDL

process can be defined nondeterministically, thus all deterministic execution paths—called interpretations—are constructed. Different interpretations are executed to obtain a set of validation candidates. Finally redundant candidates are filtered and the remaining validation fragments are send to the appropriate validators. Individual phases of the validation dispatching process are further examined in the following text.

Figure 2.2, "NVDL validation process" depicts an example validation dispatching process. A compound document with three vocabularies is used as input. In the first phase, the document is decomposed into sections depending on the namespace of the different fragments. According to the NVDL script, different actions (`validate` and `attach`) are executed on the particular sections. This leads to construction of validation candidates. Finally, for every `validate` action a non-redundant candidate is send for validation.

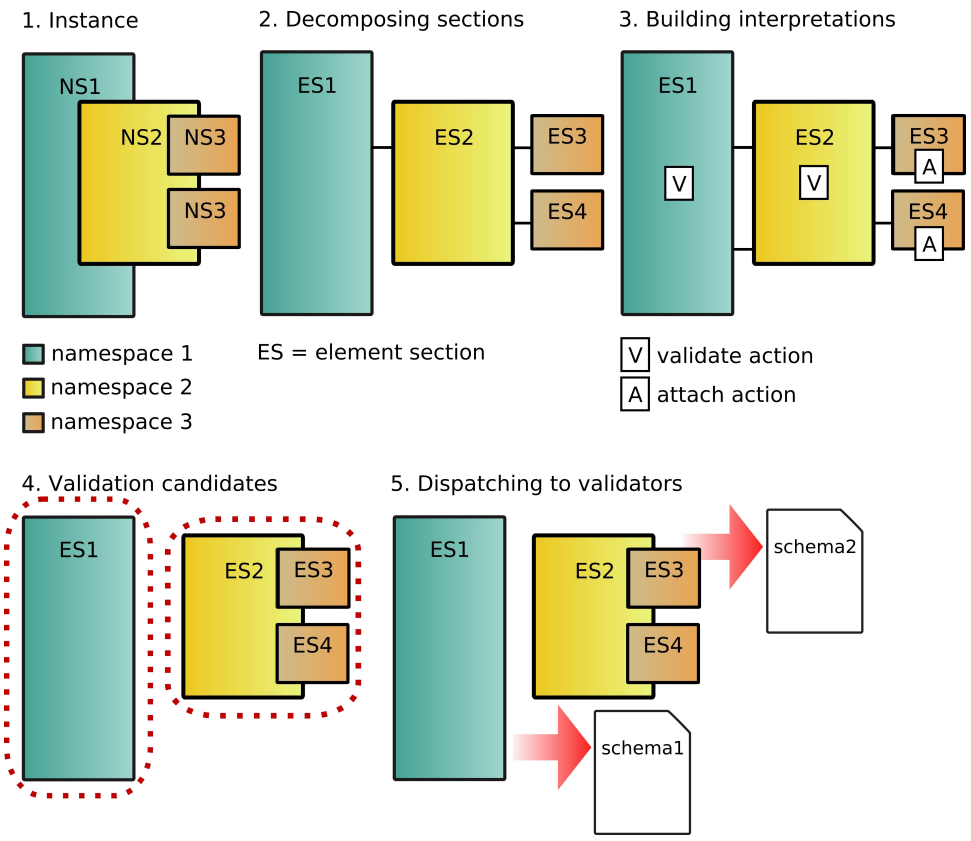

Figure 2.2. NVDL validation process

2.2.1. Data model

The first section of the specification introduces the NVDL data model. It is basically an extension of the Relax NG data model described in DSDL specification Part 2 [RNG] which defines elements and attributes as an abstract representation of the XML elements and attributes. NVDL extends elements to have element slot nodes and attribute slot nodes. Those are basically slots for decomposed element and attribute sections at their

former parent. Element section is in fact an element belonging to the same namespace as all his descendants. An attribute section is defined as a non-empty set of attributes belonging to the same namespace.

2.2.2. Syntax

There are two different NVDL syntaxes; the full syntax and the simple syntax. Both of them are formally described in the specification using Relax NG schemas. Where the full syntax is designed for maximum user experience and human readability, simple syntax is an completely internal format and should never be used by end users. The syntax is not called simple, because it is simple to understand for users. Its purpose is to simplify processing for the NVDL dispatcher.

The specification formally defines a process called simplification which transforms an NVDL schema in full syntax into simple syntax. The process is described in detail by fourteen transformation rules. The rules are ordered, but they can be executed in any order which would guarantee the same result.

Simple syntax is basically a subset of the full syntax. Here is a list of some of the most important transformation rules. For an exhaustive enumeration refer to [NVDL].

- Rules which are children of the `rules` element are wrapped into a `mode`. This `mode` is made the initial mode using the `startMode` attribute.
- Modes nested within actions are transformed to be direct children of the `rules` element and referenced using the `useMode` attribute.
- If not already present, default values are explicitly attached to rules. By default, rules match elements and the default wild-card value is "*". Any rule matching both elements and attributes is splitted into two different rules, each of which match just elements or just attributes.
- Included modes are merged with their parents. Every rule with the same condition as one of the rules within the parent is overridden by the parent rule. This is applied repeatedly until there are no included modes.
- If not already present, a default `anyNamespace` rule is added into every `mode`. As default, elements are rejected and attributes are attached.
- Every occurrence of the `allow` and `reject` elements is replaced by a `schema` element containing `allow` and `reject` elements in the `http://purl.oclc.org/dsdl/nvdl/ns/predefinedSchema/1.0` namespace. Those two elements form a simple schema language used internally by NVDL. Validation again such schemas leads just to two different results. The instance is a priori allowed, if the `allow` element is present, or rejected otherwise.

- For every action missing an `useMode` attribute, a `useMode` referencing the current `mode` is added.

The simplification process is not limited just to transformation, it also defines some constrains which shall be satisfied by a correct schema and their validation procedure. During simplification, rule collisions within modes are detected. The aim is to ensure just one rule is triggered by an namespace name, thus at most one `unwrap`, `attach` or `attachPlaceholder` action is invoked for the same section.

Example 2.17. Simplification (RDF in XHTML)

The following example shows a schema in full syntax which is used for validation of XHTML documents with embedded RDF meta-data. The same schema could have been applied in the NVDL process demonstrated in Figure 2.2, "NVDL validation process", where namespace1 corresponds to XHTML, namespace2 to RDF and the third namespace is any arbitrary foreign namespace different from namespace1 and namespace2. RDF has to be placed in the `head` context of the document. As RDF may contain fragments from arbitrary namespaces, such fragments are attached back to the RDF section.

```
<rules
     xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.rng" message="Validate against XHTML ▶
subschema.">
      <context path="/html/head" message="Allow RDF just in the context ▶
of the head element.">
        <mode>
          <namespace ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
            <validate schema="rdf.rng" message="Validate against RDF ▶
subschema.">
              <mode>
                <anyNamespace>
                  <attach message="Attach foreign elements."/>
                </anyNamespace>
              </mode>
            </validate>
          </namespace>
        </mode>
      </context>
    </validate>
  </namespace>
</rules>
```

The previous schema in full syntax is transformed by the NVDL dispatcher during the simplification process into the following schema in simple syntax. The schemas are

semantically equivalent[12]. The simplified form appears to be much longer and less readable. One of the reasons is, it contains additional default `anyNamespace` rules for every mode.

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="root">
  <mode name="root">
    <namespace wildCard="*" match="elements" ▶
ns="http://www.w3.org/1999/xhtml">
      <validate useMode="root" schema="xhtml.rng">
        <message>Validate against XHTML subschema.</message>
        <context useMode="head-context" path="/html/head">
          <message>Allow RDF just in the context of the head ▶
element.</message>
        </context>
      </validate>
    </namespace>
    <anyNamespace match="elements">
      <validate useMode="root">
        <schema>
          <predef:reject ▶
xmlns:predef="http://purl.oclc.org/dsdl/nvdl/ns/predefinedSchema/1.0" />
        </schema>
      </validate>
    </anyNamespace>
    <anyNamespace match="attributes">
      <attach useMode="root" />
    </anyNamespace>
  </mode>
  <mode name="head-context">
    <namespace wildCard="*" match="elements" ▶
ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      <validate useMode="foreign" schema="rdf.rng">
        <message>Validate against RDF subschema.</message>
      </validate>
    </namespace>
    <anyNamespace match="elements">
      <validate useMode="head-context">
        <schema>
          <predef:reject ▶
xmlns:predef="http://purl.oclc.org/dsdl/nvdl/ns/predefinedSchema/1.0" />
        </schema>
      </validate>
```

[12]The `useMode` attribute values would normally be generated automatically, thus the values would contain some sort of an auto-incremental sequence.

```
      </anyNamespace>
      <anyNamespace match="attributes">
        <attach useMode="head-context" />
      </anyNamespace>
    </mode>
    <mode name="foreign">
      <anyNamespace match="elements">
        <attach useMode="foreign">
          <message>Attach foreign elements.</message>
        </attach>
      </anyNamespace>
      <anyNamespace match="attributes">
        <attach useMode="foreign" />
      </anyNamespace>
    </mode>
  </rules>
```

2.2.3. Decomposing instances into sections

In the next phase, the specification describes the process of decomposing XML instances into attribute and element sections. For every element, attributes within the same namespace are removed and grouped into the same attribute section. For every attribute section an attribute slot node is created and attached to the parent element.

Element sections are created from elements whose parent is in a different namespace. Such element is than detached and an element slot node is create for the new section at the former parent.

Example 2.18. Decomposing XML instance into sections (RDF in XHTML)

The following example shows an HTML document with embedded RDF and Dublin Code semantics. This document has the same structure as the instance shown in Figure 2.2, “NVDL validation process”.

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Some Page</title>
    <rdf:RDF xmlns:dc="http://purl.org/dc/elements/1.1/" ▶
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
      <rdf:Description dc:title="1984 Nineteen Eighty-Four">
        <dc:creator>George Orwell</dc:creator>
        <dc:identifier>ISBN 014118776X</dc:identifier>
      </rdf:Description>
    </rdf:RDF>
</head>
<body>
```

```
    <p>It was a bright cold day in April, and the clocks were striking ▶
  thirteen...</p>
  </body>
  </html>
```

The following sections are created during section decomposition. A similar result can be observed in Figure 2.2, “NVDL validation process”.

```
  Element section 1
  <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
      <title>Some Page</title>
      (Element slot node for element section 2)
    </head>
    <body>
      <p>It was a bright cold day in April, and the clocks were striking ▶
  thirteen...</p>
    </body>
  </html>

  Element section 2
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <rdf:Description (attribute slot node for attribute section 1)>
      (Element slot node for element section 3)
      (Element slot node for element section 4)
    </rdf:Description>
  </rdf:RDF>

  Element section 3
  <dc:creator xmlns:dc="http://purl.org/dc/elements/1.1/">George ▶
  Orwell</dc:creator>

  Element section 4
  <dc:identifier xmlns:dc="http://purl.org/dc/elements/1.1/">ISBN ▶
  014118776X</dc:identifier>

  Attribute section 1
  dc:title="1984 Nineteen Eighty-Four"
```

Section decomposition doesn't need to be cause just by namespaces. With triggers a section is created for any matching element local name defined. More information about triggers can be found in Section 2.1.10, “When lacking namespaces”.

The specification also describes other manipulations with sections; e. g. re-attaching sections back to their former parent. This basically means the element slot node for the

section is replaced by its root element. Attribute sections can be reattached back to their elements in a similar way.

2.2.4. Constructing interpretations

An interpretation binds each section to exactly one action. Such action is retrieved from the sections matching rule in the current mode. Sections current mode is actually the mode there the parent element section's[13] action transits. For root element section, the current mode is the initial mode of the NVDL script. An interpretation is basically an execution plan, which tells the NVDL dispatcher what actions shall be later executed on the particular element sections.

For an deterministic NVDL schema, there is just one interpretation. But rules can have several actions, each of which can transit to different modes. Such NVDL script is nondeterministic, thus there are more different interpretations which have to be constructed. Interpretation construction for a non-deterministic NVDL schema is illustrated in Appendix B, *Interpretations for a non-deterministic NVDL Schema*.

Example 2.19. Interpretation construction (RDF in XHTML)

This example uses the instance shown in Example 2.18, "Decomposing XML instance into sections (RDF in XHTML)" and applies the NVDL script shown in Example 2.17, "Simplification (RDF in XHTML)" on it. There is just one interpretation shown in the table below. This interpretation is also depicted graphically in the Figure 2.2, "NVDL validation process" diagram.

Table 2.1. Interpretation

Section	Namespace	Mode	Action
Element section 1	XHTML	root	VALIDATE (xhtml.rng)
Element section 2	RDF	head-context	VALIDATE (rdf.rng)
Element section 3	DC	foreign	ATTACH
Element section 4	DC	foreign	ATTACH
Attribute section 1	DC	head-context	ATTACH

2.2.5. Combining section and validation

For every interpretation, the NVDL dispatcher executes all the actions assigned to different sections. Actions as `attach` or `unwrap` combine different sections into larger fragments. Those fragments are called validation candidates. The world candidate is used because they are filtered before an validate action sends them to the appropriate validator.

[13] Parent section for section `s` is a section which contains an element slot node for `s`.

Some of the validation candidates are subsets of other candidates, thus they should be filtered out. It doesn't make sense to send both such fragments for validation. Every error in the subset is apparently also detected by validating the larger fragment. Sending both fragments for validation would lead to unnecessary performance overhead and duplicate error results.

A validator is finally invoked for each remaining fragment. The appropriate validator for each schema language is determined by subschema root element namespace or from its MIME type. It can also be specified using the `schemaType` attribute.

Example 2.20. Validation candidates (RDF in XHTML)

Executing all actions in Example 2.19, "Interpretation construction (RDF in XHTML)" creates the following validation candidates. None of those candidates is filtered out, thus all candidates are send for validation. A graphical representation of the following validation fragments is shown in Figure 2.2, "NVDL validation process".

The first candidate is send for validation against the `xhtml.rng` subschema.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Some Page</title>
  </head>
  <body>
    <p>It was a bright cold day in April, and the clocks were striking ▶
thirteen...</p>
  </body>
</html>
```

The second candidate is send for validation against the `rdf.rng` subschema.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" ▶
xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description dc:title="1984 Nineteen Eighty-Four">
    <dc:creator>George Orwell</dc:creator>
    <dc:format>Book</dc:format>
    <dc:identifier>ISBN 014118776X</dc:identifier>
  </rdf:Description>
</rdf:RDF>
```

Chapter 3

JNVDL

JNVDL is an implementation of the NVDL specification (described in Chapter 2, *NVDL*) which was developed from scratch as part of this thesis. JNVDL is implemented using pure Java and it relies only on one external dependency; the JDOM library—a popular library for mapping XML instances into Java objects. Those objects can than be manipulated easily and finally outputted back into XML.

Beside JDOM, JNVDL uses just default Java 5 or later APIs and libraries. For example XML processing is done through the standard Java API JAXP 1.3. It enables applications to parse and transform XML documents independent of a particular XML processing implementation.

Part of JAXP 1.3 (newly added in Java version 5) is an integrated validation API. The API makes validation transparent to different validator implementations and it allows to invoke validators for different schema languages easily, just by specifying the language name or namespace URI. As discussed later, JNVDL makes a great use of this API and its principles.

3.1. Project and architecture

The JNVDL project consists of the core NVDL dispatcher and two other optional libraries. The NVDL dispatcher core library uses the new validation API for dispatching validation requests to different validators. One of the optional libraries is an updated version of the Kosuke's JARV to JAXP adapter[1].

JARV is an old validation API used in Java before the new JAXP API has been introduced. The adapter allows the NVDL core library to dispatch validation request to the JARV compliant Sun Multi-schema Validator (MSV), even it doesn't implement the new API. Using the adapter, it is easy to enable JNVDL to validate subschemas supported by MSV; subschemas written in several widely used schema languages e. g. Relax NG or XML Schema. The only think needed is to place the adapter's jar library and MSV library into the classpath.

[1] http://www.kohsuke.org/jarv/tiger

The second optional library is used for backward compatibility reasons. When present, NVDL validation requests can be invoked using the old JARV API.

3.1.1. Domain model

The JNVDL core NVDL dispatcher consists of four main packages: `dispatching`, `domain`, `process` and `service`.

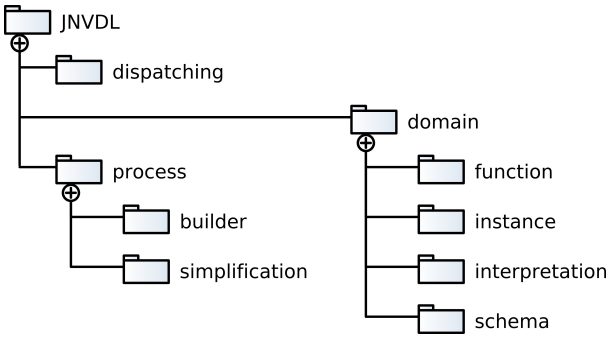

Figure 3.1. JNVDL main package structure

In the `domain` package, there are classes which represent the NVDL data model; e. g. `ElementSection`, `AttributeSection`, `ElementSlotNode` and other domain model implementations. There are several other subpackages in the `domain` package. In the `function` subpackage, there are implementations of different NVDL matching functions e. g. for matching path expressions with element section context paths or to match namespace URIs against the namespace wild-cards.

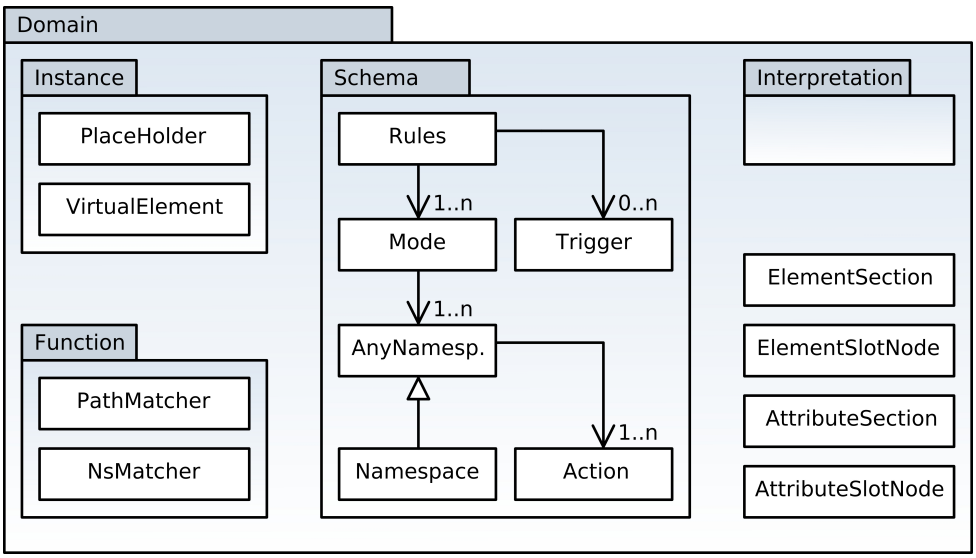

Figure 3.2. The `domain` package

Further, there is an `instance` subpackage which contains implementations of special NVDL instance elements e. g. the `placeHolder` or the `virtualElement`. In the `interpretation` subpackage, there is everything which is needed to build and access all interpretations for a particular NVDL schema and document instance pair. Classes from the `schema` subpackage are responsible for building an object representation of a particular NVDL schema. Every NVDL language element has its implementation in this package.

Finally, the `validation` package is used to embrace functionality related to validation candidates and their filtering.

3.1.2. The validation dispatching process

An important package is the `process` package, which is basically the implementation of the NVDL validation dispatching process. It involves construction and manipulation of NVDL domain model classes according to the specification.

There are two important classes in the root of the package. One of them is the `RulesCompiler` which takes the NVDL schema in XML as input and provides an `Rules` object as an entry point to the object representation of the NVDL script. Before the script is processed, it is first transformed into the simple syntax. The `Simplifier` class does this job using a transformer chain. Such chain allows to pass an input XML document along an arbitrary number of XSLT transformations, where the output of one transformation acts as an input for the next transformation in the chain.

The whole simplification process is implemented in five XSLT stylesheets which shall be applied in correct order to guarantee a correct result. Having the simplification process declared in XSLT makes the implementation better understandable to a larger group of people. XSLT also makes the implementation platform independent, as the same stylesheets may be used in other NVDL implementations independent of the platform or programming language it is based on.

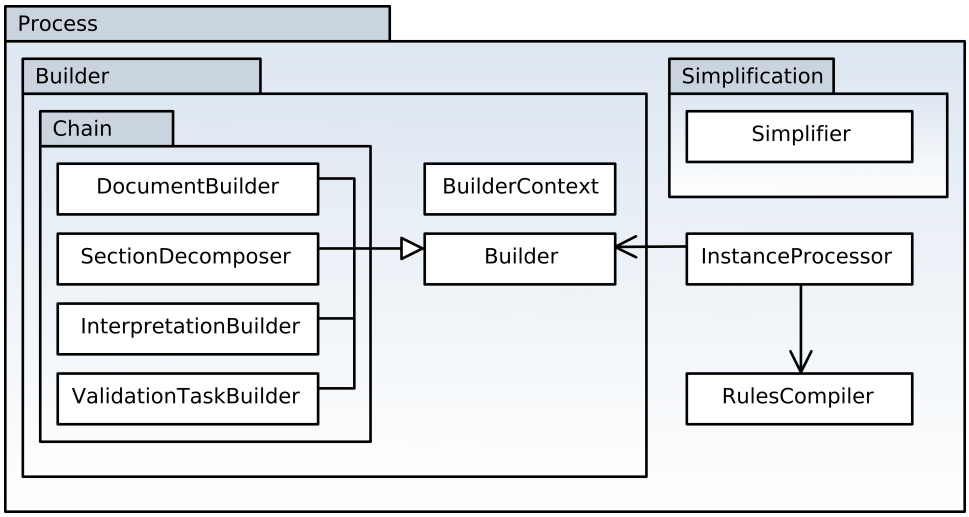

Figure 3.3. The process package

Another important class in the `process` package is the `InstanceProcessor` responsible for processing instances according to the NVDL schema. `InstanceProcessor` contains a chain of `Builders`. A `Builder` basically takes the result of the previous `Builder` in the chain, applies some manipulations to it and finally it passes the result to the next `Builder`. There are several `Builders` implemented in JNVDL and each of them corresponds to a different phase of the NVDL validation dispatching process.

Under the `builder` subpackage there is a `DocumentBuilder` which turns an XML instance into an JDOM object representation, `SectionDecomposer` which creates element and attribute sections and binds them together using element and attribute slot nodes, `InterpretationBuilder` constructs all possible interpretations and finally the `ValidationTaskBuilder` combines different sections for every interpretation by executing the appropriate actions. A set of validation candidates is produced and redundant candidates are filtered out.

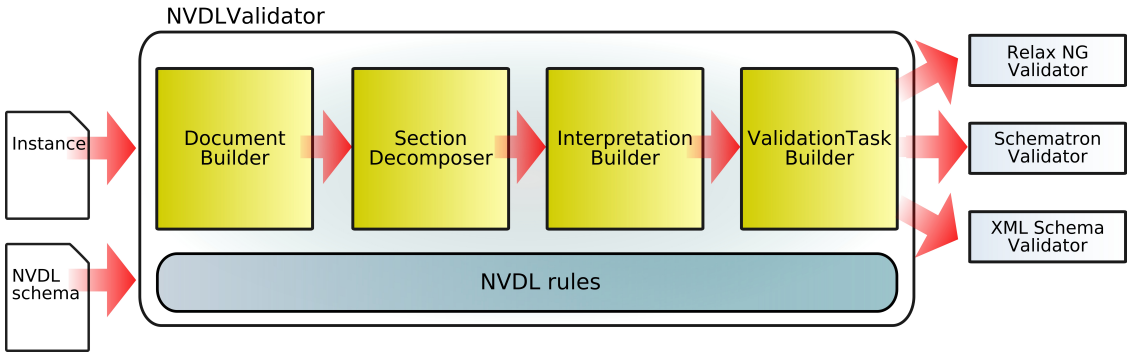

Figure 3.4. The NVDL process `Builder` chain

The complete chain takes an XML instance as input and after applying all different `Builders`, the chain returns a collection of `ValidationTask` objects. Those are basically XML fragments which shall be validated against the same schema. This collection is finally used by an `NVDLValidator` class which dispatches such fragments to the right validators using the available validation API.

NVDL validation can be directly invoked through the `NVDLValidator` class, which is based in the `dispatching` package. It contains implementations of the JAXP validation API related classes as discussed in the next section.

Additional information about the JNVDL project architecture, package structure and implementation are available in the JNVDL source code provided as part of this thesis or accessible through the project Web pages at SourceForge http://sourceforge.net/jnvdl[2].

[2] http://sourceforge.net/jnvdl

3.2. Java validation API

The Java validation API has been introduced in Java 5 as part of the JAXP interface. The validation API allows to invoke validation tasks independently of the underlying validator implementation. Switching to a different validator implementation is just matter of configuration. For example, when at some point Relax NG validation using MSV is preferred to validation using Jing, not a single line of code needs to be changed to do the transition. The only thing needed is to replace Jing in the classpath with a the MSV library.

In the validation API, there are tree main classes which serve as an entry point to invoke validation tasks. Those are the `SchemaFactory`, `Schema` and `Validator` classes. `SchemaFactory` is constructed for a specified schema language name, which is usually the namespace of the language. The factory is basically a schema compiler. It takes a particular schema definition as input to construct a `Schema` instance. A `Schema` instance is used to retrieve `Validator` instances. Where `Schema` is reused for all validation tasks against the same schema, the `Validator` instance is used for a particular validation task only.

Example 3.1. Using the Java validation API

```
Schema schema =
SchemaFactory.newInstance(schemaLanguageName)
.newSchema(new File(schemaFile));
schema.newValidator().validate(instance);
```

The question here is, how does the validation API knows, which validator implementation to invoke for a particular schema? The API uses a simple mapping between schema language names and validator implementations. When `newInstace` method is invoked at `SchemaFactory`, the validation API tries to find the appropriate validator for the specified schema language name.

To find the right `SchemaFactory` implementation for a given schema language, the following tasks are performed in order:

1. JAXP first looks into system properties to find a property of the name `javax.xml.validation.SchemaFactory:schemaLanguage`. If there is such property, its value is used as the full name of the validator class, which is than instantiated using Java reflection.

2. If there is nothing in the system properties, the next place where to look is the `$java.home/lib/jaxp.properties` file.

3. If the property is in neither of those places, the last change is to find a service provider configuration file called `javax.xml.validation.SchemaFactory` in the

resource directory (`META-INF/services`) and to look for the following mapping: `'schema language name' = 'validator class'`.

4. Finally, if no entry is found, the API throws an exception saying that no validator implementation was found for that particular schema language.

Such hierarchical approach for specifying mappings may be useful in some scenarios. For instance, if there is MSV registered in `META-INF/services` to validate Relax NG schemas, but Jing should be used instead for a particular validation. A simple solution is to specify a system property pointing to the Jing validator class. As system properties are looked up first, the MSV entry is simply overridden.

There are two implementations of the Java validation API in JNVDL. The first one is intended to process NVDL schemas. It consists of the `NVDLValidator`, which is registered to handle the NVDL schema namespace `http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0`, the `NVDLSchema` (a Java validation API `Schema` implementation) uses the `RulesCompiler` to provide an object representation of NVDL script and the `NVDLValidator` uses the `InstanceProcessor` to create validation fragments.

Resulting fragments are finally dispatched to the appropriate validators again using the Java validation API. The schema language name is extracted from the schema definition or from the `schemaType` attribute, if specified in the NVDL script. Thanks to the validation API, it is easy to incorporate a new schema language into the validation dispatching process. No special changes to JNVDL are required. The only think needed is to provide an implementation of a validator for that language and put the right library on the classpath.

The second implementation is the `PredefinedSchemaValidator` which has a very simple functionality. It validates schemas in the `http://purl.oclc.org/dsdl/nvdl/ns/predefinedSchema/1.0` namespace. The language has just two elements: `reject` and `allow`. Every time a fragment is validated against a schema which contains the `reject` element, the fragment is considered to be invalid. In case of `allow`, the fragment is valid.

3.3. Specification weaknesses

A specification as a theoretical concept does not always solve all the issues which arise when using some technology in reality. To overcome those issues, implementations may provide some proprietary functionality which goes beyond the specification without colliding with it. This is also the case for JNVDL as discussed further.

3.3.1. Round tripping

XML round tripping is a controversial issue. When parsing an XML document, in most cases, the parser is used to report just significant information; e. g. elements, attributes or processing instructions. On the other hand, there are many other information which

are semantically insignificant, but they can have a special meaning for the XML document author.

Round tripping is a big issue for XML editor developers. From the semantic point of view, an XML document with additional whitespaces inside tags is fully equivalent to the same document which doesn't have any. But from the document author perspective, those documents are not equivalent at all, because formatting documents with whitespaces may improve readability. For example breaking a long line of attributes into several lines using line breaks and tabs reduces horizontal scrolling when reading the document.

As parsers usually don't provide information about some whitespaces (e. g. whitespaces between attributes), there is no easy way to keep an XML document unchanged after being parsed and outputted back again. For example the order of attributes at an element doesn't semantically matter, but can matter for the author, an empty element can by expressed equivalently using just one tag or using a start and end tag sequence, duplicate namespace prefix declarations can be optimized etc...

XML editors have to cope with round tripping in some way. Otherwise the users would experience strange behavior as their document formatting can change every time they do save their work.

Why are actually parsers lacking round tripping support? The answer is probably performance. As round tripping would significantly slow down the parsing speed and there is just a small community of developers which really need such feature, it was apparently not the highest priority to allow round tripping even as an optional feature.

Round tripping is not just a problem for XML editors, it is also a big issue for any NVDL implementation. Validators usually tend to report error location using line and column numbers, but when parsing the instance and turning it into validation fragments, original line and column number information is inevitably lost. This makes NVDL very hard to use in the real world. It is confusing for the users to get error line numbers which aren't related to the original document line numbers at all. To interpret the information correctly, users would need to know which validation fragments were created by JNVDL and how do they look like.

This is the reason why JNVDL provides two proprietary extensions to NVDL. The first one simply preserves line numbers for elements. This feature can be turned on by setting the validator feature `http://jnvdl.sf.net/jnvdl/keep-line-numbers` to `true`. In this case, JNVDL remembers the original line number for every element and validation fragments are constructed in a way to keep elements on their original line position. For example, if some of the element sections gets unwrapped, the section won't be present in the validation fragment, but its attached child element section is going to be positioned correctly only when providing an appropriate number of empty lines previously occupied by the parent section.

Although preserving line numbers doesn't have a significant impact on performance, it is not an exhaustive solution to the problem. One of the issues is, the column numbers correctness is not guaranteed at all. In some situations even the line number information may be wrong. This can happen in case there are line breaks between individual attributes. As whitespace inside tags is semantically insignificant, parsers do not inform about it, thus the whole tag is finally outputted on one line. In case one of the attributes is invalid, the error is always reported on the line where the tag starts and not where the attribute occurs in the original document.

To solve some of the mentioned issues, JNVDL provides another proprietary extension which can be turned on by setting the validator feature `http://jnvdl.sf.net/jnvdl/round-trip` to `true`. This feature is experimental and doesn't provide an exhaustive round tripping yet. When turned on, not just line numbers of particular elements are recorded, but also whitespaces between attributes are remembered. This approach guarantees the line number information correctness. The column number information is expected to be correct in most cases.

As there is no direct support for retrieving whitespace information inside tags in the JAXP API, enabling the JNVDL round tripping feature can significantly decrease parsing performance.

The problem of irrelevance of some whitespaces within XML documents makes the use of line numbers to locate elements and attributes problematic or even error-prone. Validation API designers should consider using a different mechanism to locate errors. For example XPath is a good candidate as it is whitespace independent and precise.

Example 3.2. The round tripping issue

```
1 <root
2  attr1="foo"
3    attr2="bar">
4
5    <body></body>
6 </root>
```

The XML fragment above is semantically equivalent to the one below, but the line and column numbers for particular elements and attributes are different.

```
1 <root attr2="bar" attr1="foo">
2
3    <body/>
4 </root>
```

3.3.2. Problems with context

In the early versions of the NVDL specification, `context` handling was specified wrongly. Details are available in NVDL technical corrigenda[3] for page 20, clause 8.4. This problem was spotted thanks to the JNVDL project during the implementation phase. Later it has been corrected in collaboration with the NVDL specification authors. `Context` may override the action's default transition to point to a different mode depending on the child section's path within the parent. Detailed description of `context` semantics is described in Section 2.1.8, "Context dependent processing".

In terms of `context`, the NVDL specification was inconsistent with the provided test cases as well as with the `context` behavior described in the NRL tutorial, see [NRL]. Implementing an NVDL dispatcher precisely according to such specification would make it hard or even impossible to make any use of the `context` element at all.

The problem was caused by the way how the specification handled action transitions. An interpretation is constructed for every action in a rule triggered by some element section. If there are no `context` children, the next mode is determined using the action's default transition. Otherwise the section's context path within its parent section is used to determine the `context` to be triggered.

Before correcting the specification, current element section's[4] context path was used to determine the transition to the next mode. To handle context consistently with the test cases, the initial mode should be determined for each child section using the child section context path. Otherwise the `context` element can hardly be useful at all.

The problem can be easily demonstrated for a root element section. As the path of a root element section is an empty string, just a `context` with an empty path expression can be triggered for such section. Moreover, such context is triggered every time a rule matches a root section.

There is a simple solution to the `context` problem proposed by the author of this thesis. Initial modes for element sections should be determined when constructing an interpretation for their parent section. For every action and every child element section the action's transition shall be determined using the context path of the child section (instead of the parent context path as defined previously in NVDL).

Example 3.3. Interpretation construction with wrong context handling

In Example 2.19, "Interpretation construction (RDF in XHTML)", an interpretation is constructed for an instance with RDF vocabulary contained within XHTML and an NVDL schema which allows RDF just in context of the XHTML `head` element. The fol-

[3] http://www.jtc1sc34.org/repository/0816.htm

[4] Current element section is meant to be the section which matched the current action's rule in the current mode.

lowing interpretation is constructed for the same instance and schema, but handling context as specified in NVDL before patching the context problem.

RDF is rejected in any case, even when present directly inside the `head` element. The behavior significantly differs from the semantics of the NVDL schema.

Table 3.1. Interpretation

Section	Namespace	Mode	Action
Element Section 1	HTML	root	VALIDATE(xhtml.rng)
Element Section 2	RDF	root	REJECT
Element Section 3	DC	root	REJECT
Element Section 4	DC	root	REJECT
Element Section 5	DC	root	REJECT
Attribute Section 1	DC	root	ATTACH

3.4. Distribution and testing

Distribution related tasks in JNVDL are handled by Ant[5] which is a standard crossplatform build tool for Java projects. The Ant script, which is part of JNVDL, contains tasks for cleaning, compiling, building JAR libraries, running tests and creating test reports. Some of the tasks available are listed below.

Main JNVDL Ant tasks

clean
: Cleans the whole JNVDL project.

compile.nvdl
: Compiles the NVDL core classes to the build directory.

compile.jarv
: Compiles classes related to the JNVDL backward compatibility with the JARV validation API.

compile.bridge
: Compiles the Java validation API adapter for MSV classes.

compile.tests
: Compiles JNVDL tests.

compile
: Compiles the whole JNVDL project.

[5] http://ant.apache.org

dist.nvdl
: Creates the core JNVDL `jnvdl.jar` library.

dist.bridge
: Creates the `jarv-jaxp-bridge.jar` library to allow MSV to be accessible through the new Java validation API.

dist.jarv
: Creates the `jnvdl-jarv.jar` library to allow backward compatibility with the JARV validation API.

dist
: Creates all JNVDL JAR libraries as well as the execution environment for running NVDL validation.

test
: Runs all JNVDL functional and unit tests and generates test reports in HTML format.

Part of the JNVDL project is an extensive test suit which allows automated testing of all important JNVDL features and functionality pieces from the unit level to the functional level perspective. In the test suit, there are over sixty different tests implemented using the popular JUnit framework.

An automated test suit helps to keep the code consistent during development. Every time the code is touched, it is easy to proof all features are still functional just by executing the whole test suit again. When some changes cause some features to malfunction, with current test coverage, there is a good chance to detect such problem on the unit or functional level.

3.5. Using JNVDL

As JNVDL implements the Java validation API (and also JARV API), it can be easily used programmatically from within any Java application by invoking the API using the NVDL namespace as the schema language name parameter.

Example 3.4. Using JNVDL programmatically

```
Schema schema = ▶
SchemaFactory.newInstance("http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0").newSchema(new ▶
File("schema-file.nvdl"));
schema.newValidator().validate("instance-file.xml");
```

An alternative approach to start an NVDL validation is to use the executable scripts bundled with the JNVDL distribution. In the root of the binary JNVDL distribution, there is a `jnvdl.bat` script intended to be used in Windows NT environment and a `jnvdl.sh` for Unix systems.

Running those scripts without any parameter causes a help screen with usage information and options to be displayed. The scripts can be executed with different options. The option for specifying the NVDL schema is required, others are optional. The last parameter is the path to the validated instance. An exhaustive list of all different options follows.

JNVDL execution script options

-s
: Path to the NVDL schema. This option is required.

-rr
: `LSResourceRsolver` class to be used for resolving resources withing the instance.

-eh
: `ErrorHandler` class. The default is just outputting validation errors into the standard output. For more sophisticated behavior, a custom error handler may be used.

-er
: `EntityResoler` class to be used for resolving entities within the instance.

-d
: Enables the debug mode. In this case JNVDL outputs detailed information about the current validation dispatching process.

-D
: Means the same as the -d option, but it also enables JAXP debugging messages. This is in particular useful when debugging why JAXP cannot find a validator implementation for a particular schema language.

-f
: Specifies a validator feature to be used. The value has a name=true/false format. This option may be specified more than once.

-p
: Specifies a validator property to be used. The value has a name=value format. This option may be specified more than once.

-n
: By default, the script uses the JNVDL validator, but as it invokes the Java validation API, it can be easily used to validate against an arbitrary schema language. This parameter can change the default NVDL schema language name to a different one. For example, if `http://relaxng.org/ns/structure/1.0` is used as the parameter value, a Relax NG validator is invoked for validation.

-xi
: Toggle XInclude support in the parser. By default XInclude is turned off.

Example 3.5. JNVDL usage examples

```
The simplest usage.
jvdl.sh -s html-rdf.nvdl instance.html

Enable round-tripping.
jvdl.sh -f http://jnvdl.sf.net/jnvdl/round-trip=true -s html-rdf.nvdl ▶
instance.html

Validating against an Relax NG schema.
jvdl.sh -n http://relaxng.org/ns/structure/1.0 -s html-rdf.rng ▶
instance.html
```

Chapter 4

JNVDL integration into Relaxed

4.1. The Relaxed project

Relaxed is a Web document validation project. In the beginning, the aim of the project was to create a validation service which would overcome some of the limitations of the widely used W3C validator. The W3C validation service relies solely on DTDs to define various constraints. Such approach is lacking expressive power and namespace support. On the contrary, Relaxed uses modern expressive validation languages for describing maximum constrains to deliver comprehensive validation results to Web document authors in order to help them keep their documents as standard compliant as possible.

Part of the Relaxed project is a HTML 4.0 / XHTML 1.0 schema written from scratch using Relax NG with embedded Schematron rules. Many additional and even complicated restrictions have been expressed thanks to the powerful combination of those two languages. [HTML-VAL] contains a detailed expressivity overview of different schema languages. HTML 4.0 and XHTML 1.0 are the today's most widespread standards, but in addition, Relaxed is able to validate some of the WAI's WCAG 1.0 restrictions and there is also a basic support for validation of compound documents based on the namespace support in Relax NG. There are predefined schemas for validation of XHTML 1.0 + SVG 1.1, XHTML 1.0 + MathML 2.0 and XHTML 1.0 + MathML 2.0 + SVG 1.1 documents ready to be used.

In addition to the schemas, the Relaxed project consists also of an extensible validation engine written in Java. This validation engine allows several validator implementations to be applied to the same document, thus enabling validation against Relax NG schemas and Schematron rules at the same time. Moreover, the engine has support for doctype specific document handling, which is in particular useful to validate strict, transitional or frameset HTML documents against different schemas. Further it allows to apply specific filters to documents before validation. A filter is used to convert SGML-based HTML 4.01 documents into XML to make them validable by XML-aimed schema languages.

Relaxed validation capabilities are accessible for Web document authors through a Web-based interface ([1]). Authors may specify input documents using an URL or upload them directly to the Relaxed server. The validation process can be adjusted using several user options. For example the standard to be validated or the type of a compound document can be specified, doctype can be auto-detected or forced, error output can be brief or verbose and different error messages may be linked to the original document's source code.

An exhaustive description of all Relaxed features, project architecture and usage, schemas involved and their expressive power can be found in [RLXD] and [HTML-VAL].

4.2. Compound documents and Relaxed

Compound document support in Relaxed was based completely on the namespace support in Relax NG. This approach suffers several drawbacks as explained in Section 1.3.1, "Current schema languages". The modularity of the Relaxed HTML schema makes it quite convenient to create new compound schemas, but still it requires some effort when introducing a new vocabulary into the compound schema definitions. First of all, such vocabulary schema needs to be converted to Relax NG and consequently slightly modified to reflect the parent language schema structure.

To enhance Relaxed compound document validation features, as part of this thesis, the core Relaxed validation engine has been entirely replaced by JNVDL. Such step significantly speeds up the process of defining new compound schemas and it simplifies maintenance of the schemas present in the Relaxed schema repository.

Using JNVDL even allows Web document authors to use Relaxed to define their own simple ad-hoc NVDL scripts and use them for validation of their own custom compound documents. With NVDL, it is extremely easy to define a compound document validation process on the fly.

For example a user likes to validate XHTML with embedded SVG, MathML and RDF. In this case, there is no need to download the particular schemas, covert them to Relax NG and modify them to make them work together. The user can create such NVDL script in a matter of minutes as demonstrated in Example 4.1, "NVDL schema for XHTML+SVG+MathML+RDF". The reason is, existing schemas can be fully reused without making any changes to them.

Example 4.1. NVDL schema for XHTML+SVG+MathML+RDF

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="root">
          <mode name="root">
```

[1] http://relaxed.vse.cz

```
            <namespace ns="http://www.w3.org/1999/xhtml">
              <validate ▶
schema="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
                <context path="head" useMode="head"/>
                <context path="div|li|p...all block level elements" ▶
useMode="block_inline"/>
                <context path="a|em|span|...all inline elements" ▶
useMode="block_inline"/>
              </validate>
            </namespace>
          </mode>
          <mode name="block_inline">
            <namespace ns="http://www.w3.org/2000/svg">
              <validate ▶
schema="http://www.w3.org/TR/2002/WD-SVG11-20020108/SVG.xsd"  ▶
useMode="attach"/>
            </namespace>
            <namespace ns="http://www.w3.org/1998/Math/MathML">
              <validate ▶
schema="http://www.w3.org/Math/XMLSchema/mathml2/mathml2.xsd"  ▶
useMode="attach"/>
            </namespace>
          </mode>
          <mode namne="head">
            <namespace ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
              <validate schema="http://www.w3.org/2000/07/rdf.xsd" ▶
useMode="attach"/>
            </namespace>
          </mode>
          <mode name="attach">
            <anyNamespace><attach/></anyNamespace>
          </mode>
        </rules>
```

Example 4.1, "NVDL schema for XHTML+SVG+MathML+RDF" shows an NVDL script, which is referencing subschemas directly at their original locations using URLs. The script tells the NVDL engine that the only acceptable parent language is XHTML and other vocabularies are forbidden in that context. Plain XHTML is extracted from the validated document and send for validation against the official DTDs. RDF sections may only occur in the context of the `head` element. SVG and MathML fragments are allowed only in block and inline elements. Any foreign vocabulary contained within RDF, SVG or MathML fragments is attached to it before being send for validation. Any other vocabulary in any other context of the document is rejected.

This simple example demonstrates the power of NVDL. Modifying the NVDL script to allow any other vocabulary in some context is a simple and straightforward task. In

addition, the script contains only the required information about the compound language. Anything related to the grammar of the particular vocabularies is encapsulated in the subschemas where it really belongs. This makes NVDL schemas not only easy to design, but also easy to read and understand.

Instead of the official DTD-based schema used in the example, users may use the enhanced schema which is part of the Relaxed project. In this case they need to specify the URL to this schema using `http://<relaxed-server>/schema/web/xhtml-10-strict.rng` as the `schema` attribute value.

4.3. Further Relaxed extensions

Using JNVDL inside Relaxed was the major enhancement produced as part of this thesis, but there are several other smaller improvements as well. To begin with, Relaxed in no more bound just to HTML and to Web document validation only. When configuring Relaxed, different user-accessible pre-defined schemas may be grouped into categories. In addition to the Web document category, there is for example a DocBook category defined, featuring several DocBook related compound schemas.

Example 4.2. Schema resource configuration in Relaxed

This example shows how grouping of schema resources is done in the new version of the Relaxed project. Two groups with several schema resources are defined: the Web group and the DocBook group.

```
<resources>
  <group id="Web" name="Web Documents">
    <schema id="standalone-html" name="HTML 4.01 / XHTML 1.0 - no foreign ▶
vocabularies" url="http://relaxed.vse.cz/schema/web/xhtml-10.nvdl" ▶
schemaType="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
      Standalone HTML 4.01 / XHTML 1.0 documents with no foreign ▶
vocabularies allowed in any context.
    </schema>
    <schema id="html+svg" name="HTML 4.01 / XHTML 1.0 + SVG 1.0" ▶
url="http://relaxed.vse.cz/schema/web/xhtml-10+svg.nvdl" ▶
schemaType="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
      HTML 4.01 / XHTML 1.0 documents with embedded SVG which is allowed ▶
in any inline or block level element.
    </schema>
    ...
    more schemas
    ....
  </group>

  <group id="DocBook" name="DocBook Documents">
    <schema id="sigle-namespace-docbook" name="Docbook V5" ▶
```

```
  url="http://relaxed.vse.cz/schema/docbook/docbook.nvdl" ▶
  schemaType="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
       Standalone DocBook 5 documents with no foreign vocabularies allowed ▶
  in any context.
      </schema>
      <schema id="docbook-xforms" name="Docbook V5 + XForms" ▶
  url="http://relaxed.vse.cz/schema/docbook/docbook+xforms.nvdl" ▶
  schemaType="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
         DocBook 5 documents with embedded XFroms.
      </schema>
      ...
      more schemas
      ....
    </group>
    ...
    more groups
    ...
  </resources>
```

The previous version of Relaxed used a fixed filter chain to be applied to any input document. In the current version, filtering is done through specific parsers. They may be configured for different schema groups differently. For example the Web group uses a SGML to XML parser (based on TagSoup) to allow backward compatibility with the SGML based HTML 4.01 legacy documents, but other strictly XML-based groups do not need this kind of pre-processing.

In addition, groups can have specific options which may influence the validation process in specific ways. For example, the Web group uses a special option to attach forced doctype informations to documents. Doctype is used to determine HTML document's vocabulary mutation (strict / transitional / frameset) and thus it is important for the appropriate schema determination. In the new version, when validating a different group of documents and thus using a different parser, users can specify different validation properties to adjust the parsers bahaviour in a specific way. For example, in case of the Web group, conversion of legacy documents to XHTML can be switched on or off and an arbitrary doctype can be forced for a document instance.

Example 4.3. Specifying a different parser for the Web group

Here is an example configuration of a specific parser for a the Web group. If the `parserFactory` element is not present, `DefaultParserFactory` is used to provide JAXP default parser implementation. This example configures a different parser for the Web group. The `WebParserFactory` provides a TagSoup parser for handling legacy SGML-based HTML documents. Legacy documents are detected using doctype and MIME information.

```
<group id="Web" name="Web Documents">
    ▶
<parserFactory>edu.petrnalevka.relaxed.parser.WebParserFactory</parserFactory> ▶

   <schema id="standalone-html" name="HTML 4.01 / XHTML 1.0 - no foreign ▶
vocabularies" url="http://relaxed.vse.cz/schema/web/xhtml-10.nvdl" ▶
schemaType="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
      Standalone HTML 4.01 / XHTML 1.0 documents with no foreign ▶
vocabularies allowed in any context.
    </schema>
    ...
    more schemas
    ....
  </group>
```

In addition to predefined schema groups, Relaxed has been enhanced to support validation against user defined schemas. NVDL is an simple and easy to understand language for defining compound document validation and that's why it makes sense to enable users to define their own ad-hoc schemas to have full control over the validation dispatching process and to validate their own custom compound documents. Users may reference external subschemas or use subschemas which are part of the Relaxed schema repository available at `http://<relaxed-server>/schema/*`.

To give NVDL tenderfoots a quick start, Relaxed offers the “namespace restaurant” feature. In the namespace restaurant, users may choose vocabularies which are present in their custom compound documents from the vocabulary menu. Finally Relaxed generates a simple NVDL schema which allows and validates all the selected vocabularies in any context of the validated instance and rejects any other vocabularies. Such schema may be further edited and modified by the user before finally being used in the validation process.

Further enhancements has been made to the HTML document validation flow. Legacy document detection has been improved to use also MIME type beside doctype information. Moreover, additional parameters are passed to the XML parser to adjust processing of entities and enabling XInclude.

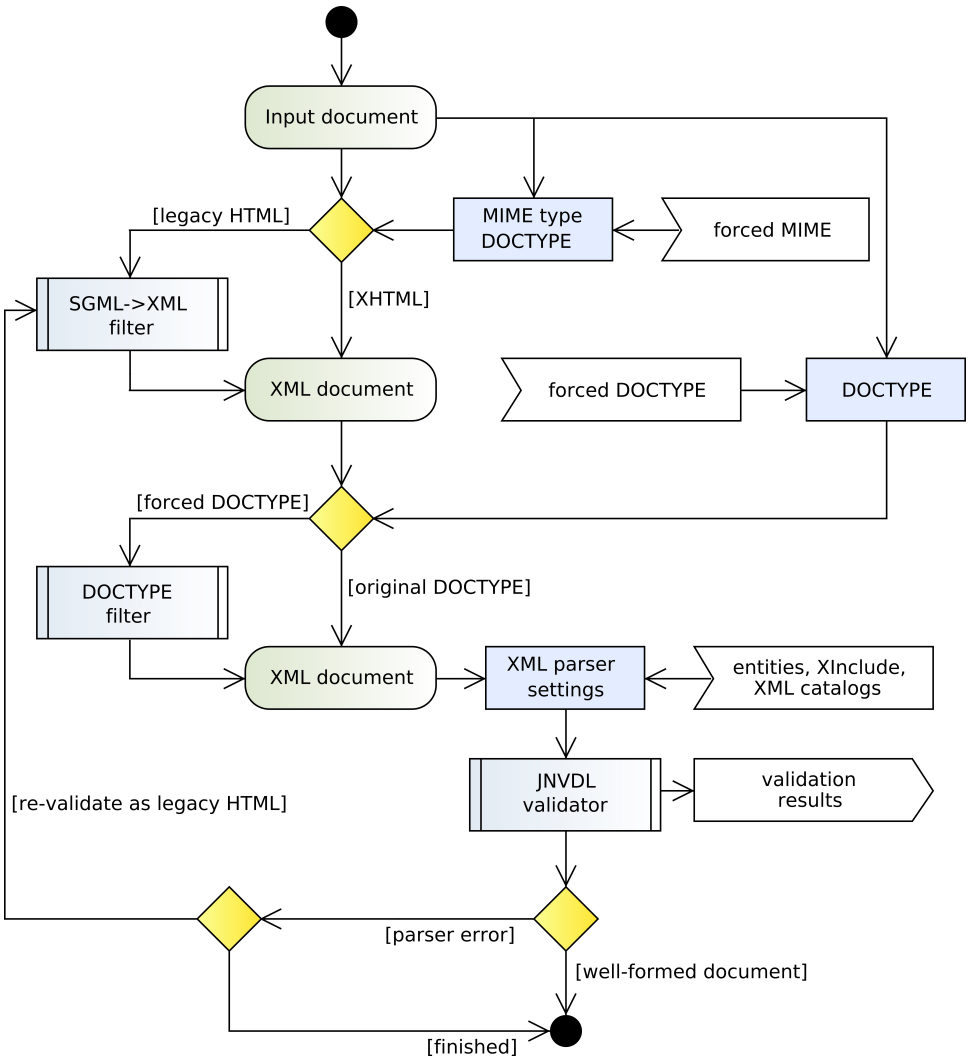

Figure 4.1. Enhanced HTML validation flow

4.4. Schematron validation

The pre-JNVDL Relaxed validation engine is able to use multiple validator implementations applied to a single instance. Error results from all validators are aggregated into a single output. Two validator implementations are used in Relaxed: a JARV validator which uses MSV to validate against Relax NG schemas and a Schematron validator able to extract Schematron rules embedded into Relax NG and perform checks against them in a separate validation process.

After replacing the old validation engine with JNVDL the above described feature has not been lost. On the contrary, NVDL also supports invocation of several validators on a single validation fragment. Such behavior is achieved using multiple `validate` actions within one rule.

Example 4.4. Multiple `validate` elements within the same rule. The validation fragment is send for validation against both, `xhtml.rng` as well as `wcag.sch`.

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.rng"/>
    <validate schema="wcag.sch"/>
  </namespace>
</rules>
```

Also the functionality of the Relaxed Schematron validator has not been lost in the new version. The validator's code has been reused and the validator now implements the Java 5 JAXP validation API and is distributed as part of the JNVDL bundle. This provides JNVDL with an out-of-the-box Schematron support.

Moreover, the validator has been enhanced to automatically detect whether the provided Schema is plain Schematron or a Relax NG schema with embedded Schematron rules. In the second case, the validator performs automatic extraction of the rules. The pre-JNVDL Relaxed version is able to validate instances against a Relax NG schema and against Schematron rules within the same schema at the same time. A similar behavior is achievable using the NVDL script in Example 4.5, “Using Relax NG Schematron extraction with NVDL”.

Example 4.5. Using Relax NG Schematron extraction with NVDL

The `xhtml.rng` in this example is a Relax NG schema with embedded Schematron rules. An XHTML validation fragment is first send for validation through JAXP to a registered Relax NG validator. The second `validate` element uses the same schema for validation, but the `schemaType` attribute value forces JNVDL to use a Schematron validator instead. The validator implementation first detects from the namespace of the root element that the schema is written in Relax NG. After that, it applies a special transformation to it which extracts Schematron rules form within the schema. Those rules are finally used for validation.

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.rng"/>
    <validate schemaType="http://purl.oclc.org/dsdl/schematron" ▶
schema="xhtml.rng"/>
  </namespace>
</rules>
```

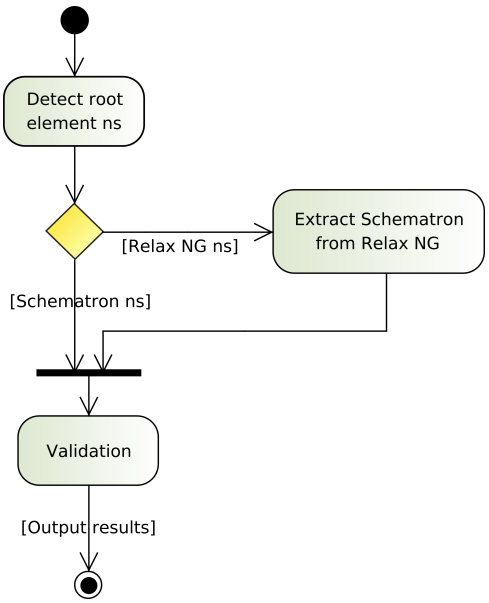

Figure 4.2. Schematron validation in JNVDL

Relaxed Schematron validator supports both, the older Schematron 1.5 which is based in the `http://www.ascc.net/xml/schematron` namespace, but also the new ISO Schematron (an international standard) with the `http://purl.oclc.org/dsdl/schematron` namespace.

4.5. New user interface

This sections provides a short introduction into the enhanced user interface of the next generation Relaxed validation service. Screenshots from the running prototype deployment are used to demonstrate different features.

Figure 4.3, “Web group” shows content of the `Web` tab which corresponds to the Web group defined within Relaxed schema resource configuration. There are other tabs available for accessing other configured groups; the SemanticWeb, XSLT or Docbook group.

In Figure 4.3, “Web group”, a test document at `http://nalevka.com/test.html` is being validated against a plain HTML 4.01 / XHTML 1.0 schema selected in the `Schema` select box. This control contains also other schemas available in the Web group.

At the bottom of the screen there are validation results. The output informs the user about what doctype has been automatically resolved for his document. One error has been detected in the validated instance, thus the document is considered to be invalid. The error message shows line and column number to locate the error in the document's source and the related source code snippet.

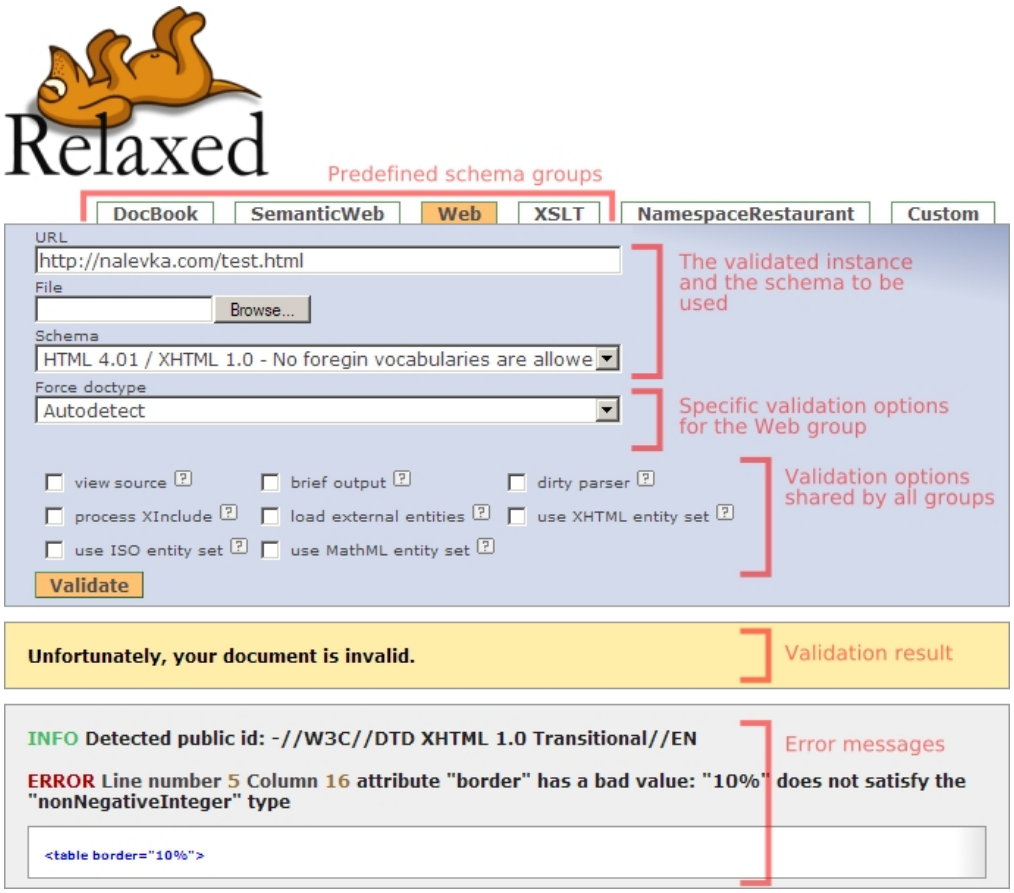

Figure 4.3. Web group

Newly, Relaxed users are not limited to the predefined schemas in the repository. They may also validate instances using their own custom schemas. Figure 4.4, “Using custom schemas” reveals the user interface for such tasks.

The schema source code is editable in the text-area. By default, schema type is set to the NVDL namespace, but users may use any other schema language supported by Relaxed. Currently there is support for DTD, Relax NG, Relax Core, Relax Namespace, XML Schema, Trex, Schematron 1.5, ISO Schematron and Schematron embedded in Relax NG.

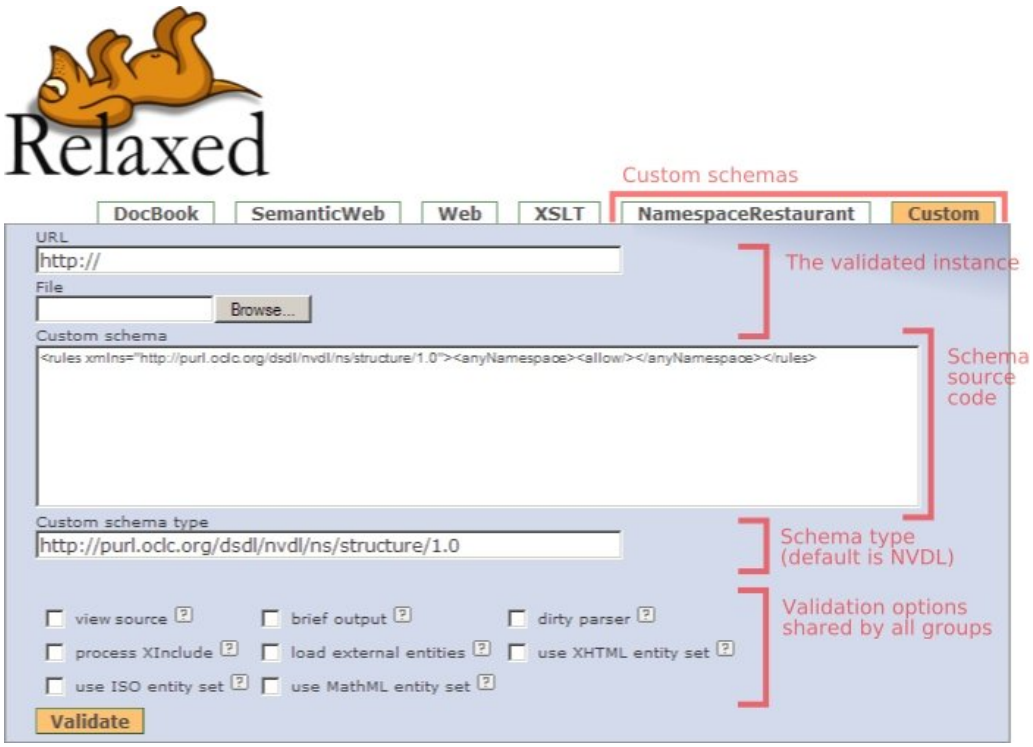

Figure 4.4. Using custom schemas

Finally, Figure 4.5, "Relaxed namespace restaurant" demonstrates the Namespace restaurant feature. Namespace restaurant user interface is very similar to the custom schema interface, but in addition it has a `Namespace menu` and the `Order` button. Also it doesn't allow to choose an arbitrary schema language, but the schema is always considered to be NVDL.

Namespace restaurant helps users to generate custom NVDL schemas. Choosing pre-configured namespaces from the `Namespace menu` and clicking `Order` results in a flat NVDL schema being generated into the schema source text-area. In Figure 4.5, "Relaxed namespace restaurant", XHTML and XForms namespace has been selected from the menu, thus the generated NVDL schema allows XHTML and XFroms in any context of the validated documents and it validates them against the appropriate schemas in the Relaxed schema repository.

Generated schemas are very simple and it is expected they give users just a quick start. Users may edit the generated source code and enhance the schema in any way they like. For example in the XHTML + XForms case, users may want to allow only XHTML as the parent language.

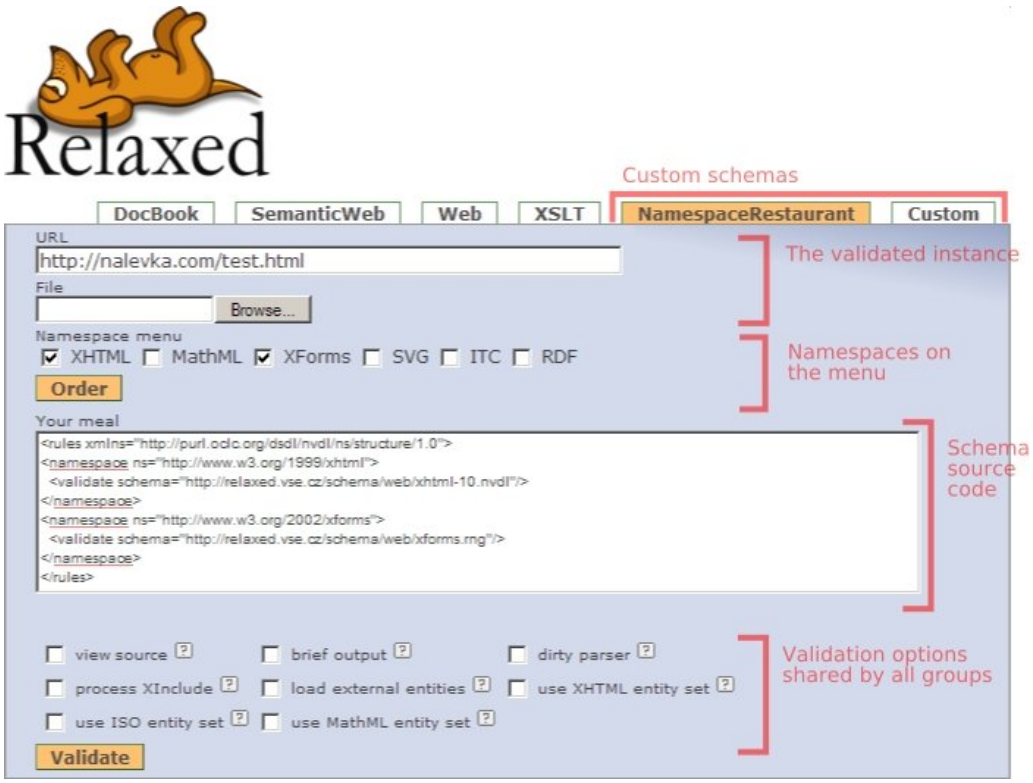

Figure 4.5. Relaxed namespace restaurant

The next generation Relaxed validation service prototype is accessible for public use and testing on the Web[2].

[2] http://relaxed.vse.cz/nextgeneration

Chapter 5

Conclusion

This thesis demonstrated that compound document validation using the traditional approach—namespace support in widely used schema languages—suffers several significant drawbacks. Creating and maintaining such compound schemas is time consuming, error-prone and requires additional knowledge. A solution to those problems proposed in this text is NVDL; an international standard for compound document validation. The NVDL concept is based on the idea of separation compound documents into single namespace fragments which are later independently send for validation against different schemas. Such approach allows to fully reuse existing single namespace schemas and it separates compound language grammar definition from the grammar definition of the individual vocabularies. Moreover, a mix of different schema languages may be used withing one validation process.

Another aim of this thesis is to illustrate how are compound documents becoming increasingly important not only in the Web environment, but also in other areas of information technologies. Ever growing significance of compound documents requires reliable and convenient validation techniques. Validation is absolutely essential to ensure interoperability of different systems which use compound documents for data exchange. That's the reason why a full-featured compound document validation environment (Relaxed) has been developed and made accessible to the wide public as part of this thesis.

This environment is based on JNVDL; the first Java implementation of the NVDL specification created from scratch by the author of this text. JNVDL validation features are made accessible through a Web based graphical user interface to compound document authors as part of the Relaxed project. Authors may freely check their compound documents and thus make them more standard-compliant and interoperable. Relaxed and JNVDL are both open source projects and their implementation is a significant contribution of this thesis.

This thesis also contributed to the NVDL international standard. During the implementation of JNVDL an error regarding context handling has been detected in the NVDL specification. Further details are available in NVDL technical corrigenda[1]; page 20, clause 8.4.

[1] http://www.jtc1sc34.org/repository/0816.htm

Moreover, this text may server as an NVDL specification reference and tutorial and it may give a quick start to anyone interested in using NVDL. All aspects of the validation dispatching process are briefly described and the NVDL language is examined in detail using illustrative examples. In addition, this thesis may serve as a high level description of the JNVDL implementation and together with other texts e. g. [RLXD] and [HTML-VAL] also for the Relaxed project. As such, it may help to get new people involved in further development and maintenance of those projects.

This thesis leaves a wide area open for further extensions and enhancements of both projects; JNVDL and Relaxed. There is a huge number of different compound languages which could be formalized in NVDL and made part of the Relaxed predefined schema repository to enable their out-of-the-box validation. Relaxed schema repository could be extended to contain additional schema groups to cover other areas of compound document usage e. g. SOAP, JSPs. Relaxed GUI could be enhanced for better user experience; making it easy to use and providing additional features as e. g. validation of several linked Web documents in one process or graphical interface for easier NVDL editing. A searchable user-maintained schema repository could be added to the Relaxed interface as well; allowing the Relaxed community to maintain and share their own compound document schemas. Relaxed validation error output could be made more verbose and explanatory by implementing annotation support into the validators used and by annotating schemas in the repository.

Further, JNVDL could be enhanced to bring a complete solution to line number location issues in error messages. Either by implementing full round-tripping support, or by enhancing subschema validators used within JNVDL to use XPath as the preferred error location mechanism. Additional extensions could be made to JNVDL and the NVDL specification to provide fine-grained rule's condition triggering or to enable the use of XPath to define sections' context. For that purpose an extension API could be made available within JNVDL to allow straightforward third-party plug-in development. JNVDL could be made more memory efficient by enabling it to operate in streaming mode, thus allowing it to process huge input documents.

References

[XML] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C, 2006. Available at: http://www.w3.org/TR/2006/REC-xml-20060816

[NS] Bray, T., Hollander, D., Layman, A., Tobin, R.: Namespaces in XML 1.0 (Second Edition). W3C, 2006. Available at: http://www.w3.org/TR/REC-xml-names

[NVDL] Murata, M.: Document Schema Definition Languages (DSDL) — Part 4: Namespace-based Validation Dispatching Language (NVDL). ISO/IEC FDIS 19757-4, 2005. Available at: http://www.jtc1sc34.org/repository/0694.pdf

[NRL] Clark, J.: Namespace Routing Language (NRL). Thai Open Source Software Center Ltd, 2003. Available at: http://www.thaiopensource.com/relaxng/nrl.html

[RLXD] Kosek, J., Nálevka, P.: Relaxed — on the Way Towards True Validation of Compound Documents. WWW2006, Edinburg, 2006. Available at: http://www2006.org/programme/files/pdf/4508.pdf

[HTML-VAL] Nálevka, P.: Doplňková validate HTML a XHTML dokumentů. University of Economics, Prague, 2003. Available at: http://nalevka.com/resources/thesis.pdf

[MNS] Clark, J.: Modular Namespaces (MNS). Thai Open Source Software Center Ltd, 2003. Available at: http://www.thaiopensource.com/relaxng/mns.html

[NSSB] Jelliffe, R.: Namespace Switchboard. Topologi pty. Ltd, 2003. Available at: http://www.topologi.com/resources/NamespaceSwitchboard.html

[RNS] Murata, M.: Regular Language Description for XML (RELAX) — Part 2: RELAX Namespace. ISO/IEC, 2001. Available at: http://www.y-adagio.com/public/standards/iso_tr_relax_ns/dtr_22250-2.doc

[RFC3986] Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax. Network Working Group, 1998. Available at: http://www.ietf.org/rfc/rfc3986.txt

[RNG] Clark, J., Murata, M.: RELAX NG Specification. OASIS Committee Specification, 2001. Available at: http://www.relaxng.org/spec-20011203.html

[SCH] Jelliffe, R.: The Schematron Assertion Language 1.5. Academia Sinica Computing Centre, 2002. Available at: http://xml.ascc.net/resource/schematron/Schematron2000.html

[XMLSCH-ST] Thompson, H., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures Second Edition. W3C, 2004. Available at: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/

[XMLSCH-DT] Biron, P., Malhotra, A.: XML Schema Part 2: Datatypes Second Edition. W3C, 2004. Available at: http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[XML-DTD] Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Third Edition). W3C, 2004. Available at: http://www.w3.org/TR/2004/REC-xml-20040204/

[XSLT] Clark, J.: XSL Transformations (XSLT) Version 1.0. W3C, 1999. Available at: http://www.w3.org/TR/xslt

[HTML4] Ragget, D., Le Hors, A., Jacobs, I.: HTML 4.01 Specification. W3C, 1999. Available at: http://www.w3.org/TR/1999/REC-html401-19991224/

[XHTML1] XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition). W3C, 2002. Available at: http://www.w3.org/TR/2002/REC-xhtml1-20020801/

[XHTML-MOD] Altheim, M., McCarron, S., Boumphrey, F., Dooley, S., Schnitzenbaumer, S., Wugofski, T.: Modularization of XHTML™. W3C, 2001. Available at: http://www.w3.org/TR/2001/REC-xhtml-modularization-20010410/

[XHTML11] Altheim, M., McCarron, S.: XHTML™ 1.1 – Module-based XHTML. W3C, 2001. Available at: http://www.w3.org/TR/2001/REC-xhtml11-20010531/

[WCAG] Chisholm, W., Vanderheiden, G., Jacobs, I.: Web Content Accessibility Guidelines 1.0. W3C WAI, 1999. Available at: http://www.w3.org/TR/1999/WAI-WEBCONTENT-19990505/

[XINCLD] Marsh, J., Orchard, D.: XML Inclusions (XInclude) Version 1.0. W3C, 2004. Available at: http://www.w3.org/TR/xinclude

[RDF] Beckett, D.: RDF/XML Syntax Specification (Revised). W3C, 2004. Available at: http://www.w3.org/TR/rdf-syntax-grammar

[SVG] Ferraiolo, J., Fujisawa, S., Jackson, J.: Scalable Vector Graphics (SVG) 1.1 Specification. W3C, 2003. Available at: http://www.w3.org/TR/SVG11

[MTHML] Carlisle, D., Ion, P., Miner, R., Poppelier, N.: Mathematical Markup Language (MathML) Version 2.0 (Second Edition). W3C, 2003. Available at: http://www.w3.org/TR/MathML2

[XLNK] DeRose, S., Maler, E., Orchard, D.: XML Linking Language (XLink) Version 1.0. W3C, 2001. Available at: http://www.w3.org/TR/xlink

Definitions

A

Ant — Ant is a platform independent build tool written in Java. Today Ant is basically a standard approach to Java project maintenance.

Attribute — In this text, the word attribute refers to an XML attribute. Attributes describe additional information about an XML element. The syntax is `attributeName="value"`. Several attributes can be present at one element, but their names has to be unique.
See Also Element, Tag, XML.

Attribute Section — Part of the NVDL data model. An attribute section is a non-empty set of attributes having the same namespace name.
See Also Attribute, Element Section, NVDL.

Attribute Slot Node — Part of the NVDL data model. An attribute slot node is a slot for an attribute section at an element.
See Also Element, Attribute Section, Element Slot Node, NVDL.

B

Browser — Software program used to view and interact with various types of resources available on the Web.
See Also HTML.

C

CSS — Cascading Style Sheets – a styling language which allows controlling appearance (fonts, colors, margins, layout...) of structured documents (for example XML documents).
See Also XML.

D

DTD — *Document Type Definition* – a simple schema language with a limited expressive power. Used to describe structure of SGML as well as XML documents. DTD doesn't feature namespace support and it doesn't have a XML-based syntax.
See Also XML, SGML.

E

Element — In this text, the word element refers to an XML element, which is the essential building block of any XML document. An element may contain textual information of other child elements. The content of an element is marked by the element's start and end tag.
See Also Attribute, Tag, XML.

Element Section — Part of the NVDL data model. An element section is basically an element such that a single namespace name applies to itself and all descendant elements.
See Also Element, Attribute Section, NVDL.

Element Slot Node — Part of the NVDL data model. An element slot node is a slot for an element section at an element.
See Also Element, Element Section, Attribute Slot Node, NVDL.

H

HTML — Hyper Text Markup Language – a mark-up language for creating documents or distributed applications on the Web. It is a SGML of XML application. In case of XML it's usually referenced to as XHTML.
See Also XML, SGML.

J

JARV — Java Application Programming Interface for RELAX Verifiers – a validation API used in Java 4 and earlier. The interface allows to invoke validation processes independently on the validator implementation and the schema language used.
See Also Schema language, JAXP Validation API.

JAXP
: Java API for XML Processing ⊣ one of the Java XML programming APIs. It provides the capability of validating, transforming and parsing XML documents independently on the underlaying validator, parser and transformer implementations.
See Also JAXP Validation API, XML.

JAXP Validation API
: A replacement for the JARV Validation API. It is implicitly bundled with Java 5 and later. The interface allows to invoke validation processes independently on the validator implementation and the schema language used.
See Also JAXP, JARV, Schema language.

JDOM
: Java Document Object Model – parses, manipulates, and outputs XML using standard Java constructs. The API is similar to DOM, but easier to use.

Jing
: Jing is a Relax NG validator written in Java with experimental support for other schema languages e. g. Schematron, NRL and XML Schema.
See Also Relax NG, Schema language, XML Schema, XML, XML Schema.

JNVDL
: Java Namespace-based Validation Dispatching Language Implementation – Java-based implementation of the NVDL specification, created as part of this thesis.
See Also NVDL.

JSP
: Java Server Pages – a scripting language intended for developing dynamic Web pages in Java. JSPs may be expressed using XML.
See Also XML.

JUnit
: A popular unit test framework for the Java programming language.

M

MathML
: Mathematical Markup Language – an XML-based language used for displaying mathematical notation and content.
See Also XML.

MSV
: Sun Multi-Schema XML Validator – a Java validation tool intended to validate XML documents against several kinds of XML schema languages. It supports RELAX NG, RELAX Namespace, RELAX Core, TREX, XML DTDs, and a subset of XML Schema Part 1.

See Also XML, Relax NG, XML Schema, Schema language.

N

Namespace

A namespace is where a set of names reside. Since they are all in the same set, their names have to be unique. In XML namespaces provide a method for qualifying element and attribute names by associating them with namespaces identified by an URI. Namespaces effectively allow the use of multiple vocabularies in one XML document.
See Also URI, XML.

NVDL

Namespace-based Validation Dispatching Language –a compound document validation approach and a "meta-schema" language at the same time. NVDL is based on the idea of separating compound documents into sections according to their namespaces, building larger fragments from that sections according to the NVDL schema definition and finally validate those fragments separately against different schemas (which can be written in different schema languages), using different validators.
See Also Attribute Section, Attribute Section, Element Section, Element Slot Node.

P

Parser

In this text, the word parser refers to an XML parser. An XML parser is a processor that reads an XML document and determines its structure and data.
See Also XML, Element, Attribute.

R

RDF

Resource Description Framework – a standard framework for describing and interchanging metadata. The simple format of resources, properties, and statements allows RDF to describe robust meta-data, such as ontological structures. RDF has an XML syntax.
See Also XML.

Relax NG

Relax NG is a simple yet elegant schema language for XML, based on Murata Makoto's RELAX and James Clark's TREX. A RELAX NG schema specifies a pattern for the structure and content of an XML document. A RELAX NG schema uses an XML syntax or a non-XML compact syntax.

See Also XML, Schema language.

Relaxed — Relaxed is a project for advanced validation of Web documents. It uses modern validation approaches to maximize validation results and it features compound document validation support.

Round-tripping — Round-tripping is conversion of a document from one format to another and than back again. Such process may cause information loss and the resulting document may not be absolutely the same after such back and forth conversion. This problem occurs when processing XML. As parsers don't provide some semantically insignificant information (e. g. some whitespace information), when outputting an XML document back to file, this kind information may be lost.
See Also XML, Parser.

S

Schema language — A language for defining elements, structure and rules an XML document must satisfy to be valid. There are two different kinds of schema languages: grammar-based, which are elegant to specify a complete grammar of a language as it is very easy to express parent-child relationships, and rule based, which are most powerful in combination with grammar-based languages to express additional complex structural rules.
See Also DTD.

Schematron — A rule-based XML Schema language, developed by Rick Jelliffe, originally using XPath expressions to describe validation rules. Although the language is very simple, it's able to express very complex structural constrains, which are hardly expressible using grammar-based schema languages.
See Also XML, XPath, Schema language.

SGML — Standard Generalized Markup Language – a standard for creating markup languages. It provides a complex set of rules for defining document structures. XML is a subset of SGML.
See Also HTML, XML.

SOAP — Simple Object Access Protocol – a XML-based protocol for exchanging structured XML messages. In SOAP,XML queries are sent to retrieve XML responses. In general, those

	are called SOAP messages, which are basically XML data wrapped up in a SOAP envelope. See Also XML.
SVG	*Scalable Vector Graphics* – a powerful language for describing two-dimensional vector graphics in XML. See Also XML.

T

Tag	a tag is a marker embedded in a document. Each element has a beginning tag and an end tag. See Also Element.

U

URI	*Uniform Resource Identifier* – a formatted string that serves as an identifier for a resource.

W

WCAG	Web Content Accessibility Guidelines – a set of recommendations aimed to make Web content accessible to people with all sorts of disabilities. See Also HTML.
Web	Short for World Wide Web – one of the Internet services. Web resources (Web pages) are identified using URIs and served by Web server using the HTTP protocol. Clients can display Web pages using a Web browser. See Also Browser.
Well-formed	An XML document is considered to be well-formed if it meets the conditions defined in the XML specification. Only a well-formed document can be processed with a XML parser without errors. See Also XML, Parser.

X

XML	eXtensible Markup Language – a simple and flexible text format derived from SGML. XML was designed for electronic publishing, but it also plays an important role as an universal data exchange format in various areas. See Also Attribute, Element, Parser, SGML.

XML Schema
: A grammar-based complex an heavy-weight standard schema language. The specification consists of two parts, one about defining structure relationships and the other about data types for constraining element and attribute values.
See Also XML, Schema language.

XPath
: *XML Path Language* – a query language used to identify a set of nodes within an XML document. It also provides basic facilities for manipulation of the nodes and their data.
See Also XML, XSLT.

XSLT
: *eXtensible Stylesheet Language Transformations* – a templating language, which can express rules for transforming a source XML tree into a result tree. The transformation is done by associating patterns with templates. Where patterns match parts of the source tree (using XPath), templates create the corresponding result. With XSLT structure of the source tree can be completely changed in the result.
See Also XML, XPath.

Z

Tag
: In this text, the word tag refers to an markup tag. A tag is basically a marker which defines content of an element. Elements have a beginning tag and an end tag.
See Also Element, XML, SGML.

Appendix A

NVDL validation dispatching process

In the beginning of Chapter 2, *NVDL*, individual phases of the NVDL validation dispatching process where demonstrated using the Figure 2.1, "NVDL validation process at a glance" diagram. This section provides an instance and a schema example which could be used during this particular validation process. The instance is further decomposed into sections and finally combined into validation fragments intended to be send for validation.

Example A.1. An instance and a schema

The following example shows an NVDL schema and a compound document instance which relates to Figure 2.1, "NVDL validation process at a glance". In this case, NS1 represents the XHTML namespace and NS2 stands for the XForms namespace.

```
<html xmlns="http://www.w3.org/1999/xhtml" ▶
xmlns:xf="http://www.w3.org/2002/xforms">
<head>
<xf:model>
  <xf:instance>...</xf:instance>
  <xf:submission id="form" method="post"  action="getStockQuote.do"/>
</xf:model>
</head>

<body>
<xf:group ref="stockquote">
<xf:input ref="symbol"><xf:label>Symbol</xf:label></xf:input>
<br />
<xf:submit submission="form"><xf:label>Get Quote</xf:label></xf:submit>
</xf:group>
</body></html>
```

To achieve behavior consistent with Figure 2.1, "NVDL validation process at a glance", the following schema is applied to the previous compound document instance. Using such schema, the NVDL dispatcher first sends the root XHTML fragment for validation after filtering any descendant XForms fragments and attaching any descend-

ant XHTML fragments. XForms sections are handled in a similar way by filtering any descendant XHTML. For any XHTML document instance with embedded XForms, the following NVDL schema causes one pure XHTML fragment to be send for validation against `xhtml.xsd` and one or more pure XForms fragments to be validated using `xforms.rng`.

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
<namespace ns="http://www.w3.org/1999/xhtml">
  <validate schema="xhtml.xsd">
    <mode>
      <namespace ns="http://www.w3.org/2002/xforms">
        <validate schema="xforms.rng">
          <mode>
            <namespace ▶
ns="http://www.w3.org/2002/xforms"><attach/></namespace>
            <namespace ▶
ns="http://www.w3.org/1999/xhtml"><unwrap/></namespace>
          </mode>
        </validate>
        <unwrap>
          <mode>
            <namespace ▶
ns="http://www.w3.org/2002/xforms"><unwrap/></namespace>
            <namespace ▶
ns="http://www.w3.org/1999/xhtml"><attach/></namespace>
          </mode>
        </unwrap>
      </namespace>
    </mode>
  </validate>
</namespace>
</rules>
```

Example A.2. Decomposing sections

The instance shown in Example A.1, “An instance and a schema” is decomposed into the following section tree.

```
Element Section 1
<html>
  <head>
    (Element slot node for element section 2)
  </head>
  <body>
    (Element slot node for element section 4)
  </body>
```

```
</html>

Element Section 2
<xf:model>
  <xf:instance>...</xf:instance>
  <xf:submission id="form" method="post"  action="getStockQuote.do"/>
</xf:model>

Element Section 3
<br />

Element Section 4
<xf:group ref="stockquote">
  <xf:input ref="symbol"><xf:label>Symbol</xf:label></xf:input>
    (Element slot node for element section 3)
  <xf:submit submission="form"><xf:label>Get Quote</xf:label></xf:submit>
</xf:group>
```

Example A.3. Dispatching validation fragments to validators

After executing `attach` and `unwrap` actions on the section tree shown in Example A.2, "Decomposing sections", the following fragments are filtered out of validation candidates and send independently for validation.

```
<html>
<head></head>
<body>
<br />
</body>
</html> -> xhtml.xsd

<xf:model>
  <xf:instance><stockquote><symbol/></stockquote></xf:instance>
  <xf:submission id="form" method="post"  action="getStockQuote.do"/>
</xf:model> -> xforms.rng

<xf:group ref="stockquote">
  <xf:input ref="symbol"><xf:label>Symbol</xf:label></xf:input>
  <xf:submit submission="form"><xf:label>Get Quote</xf:label></xf:submit>
</xf:group> -> xforms.rng
```

Appendix B

Interpretations for a non-deterministic NVDL Schema

Appendix A, *NVDL validation dispatching process* shows an example of a simple non-deterministic NVDL validation dispatching process. Multiple interpretations has to be constructed within such process. In this scenario non-determinism is cause by two different actions being executed on the same section, where both of them transit to a different mode. The following example shows all constructed interpretations. The NVDL schema in Example A.1, "An instance and a schema" has been modified to use named modes to allow referencing modes from within the interpretation table. The modified schema is in Example B.1, "NVDL schema (XFroms in XHTML) with named modes".

Example B.1. NVDL schema (XFroms in XHTML) with named modes

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0" ▶
startMode="html">
<mode name="html">
  <namespace ns="http://www.w3.org/1999/xhtml">
    <validate schema="xhtml.xsd" useMode="xforms"/>
  </namespace>
</mode>
<mode name="xforms">
  <namespace ns="http://www.w3.org/2002/xforms">
    <validate schema="xforms.rng" useMode="unwrap-html"/>
    <unwrap useMode="unwrap-xforms"/>
  </namespace>
</mode>
<mode name="unwrap-html">
  <namespace ns="http://www.w3.org/2002/xforms"><attach/></namespace>
  <namespace ns="http://www.w3.org/1999/xhtml"><unwrap/></namespace>
</mode>
<mode name="unwrap-xforms">
  <namespace ns="http://www.w3.org/2002/xforms"><unwrap/></namespace>
  <namespace ns="http://www.w3.org/1999/xhtml"><attach/></namespace>
```

```
</mode>
</rules>
```

Example B.2. Interpretation construction (XFroms in XHTML)

Table B.1. Interpretation 1

Section	Namespace	Mode	Action
Element section 1	XHTML	html	VALIDATE (xhtml.xsd)
Element section 2	XForms	xforms	VALIDATE (xfroms.rng)
Element section 3	XHTML	unwrap-html	UNWRAP
Element section 4	XFroms	xforms	VALIDATE (xfroms.rng)

Table B.2. Interpretation 2

Section	Namespace	Mode	Action
Element section 1	XHTML	html	VALIDATE (xhtml.xsd)
Element section 2	XForms	xforms	UNWRAP
Element section 3	XHTML	attach-html	ATTACH
Element section 4	XFroms	xforms	UNWRAP

Table B.3. Interpretation 3

Section	Namespace	Mode	Action
Element section 1	XHTML	html	VALIDATE (xhtml.xsd)
Element section 2	XForms	xforms	VALIDATE (xfroms.rng)
Element section 3	XHTML	attach-html	ATTACH
Element section 4	XFroms	xforms	UNWRAP

Table B.4. Interpretation 4

Section	Namespace	Mode	Action
Element section 1	XHTML	html	VALIDATE (xhtml.xsd)
Element section 2	XForms	xforms	UNWRAP
Element section 3	XHTML	unwrap-html	UNWRAP
Element section 4	XFroms	xforms	VALIDATE (xfroms.rng)

All four previously demonstrated interpretations are executed on the section tree. This results in creation of eight validation candidates which are further filtered for redundancy. Finally just three remaining fragments are send for validation; one XHTML fragment and two XForms fragments.

```
Interpretation 1
<html> (redundant, filtered out)
<head></head>
```

```
<body></body>
</html>

<xf:model>
  <xf:instance>...</xf:instance>
  <xf:submission id="form" method="post"  action="getStockQuote.do"/>
</xf:model> -> xforms.rng

<xf:group ref="stockquote">
  <xf:input ref="symbol"><xf:label>Symbol</xf:label></xf:input>
  <xf:submit submission="form"><xf:label>Get Quote</xf:label></xf:submit>
</xf:group> -> xforms.rng

Interpretation 2
<html> (Overriddes fragment in Interpretation 1)
<head></head>
<body>
<br/>
</body>
</html> -> xhtml.xsd

Interpretation 3
<html> (redundant, filtered out)
<head></head>
<body>
<br/>
</body>
</html>

<xf:model> (redundant, filtered out)
  <xf:instance>...</xf:instance>
  <xf:submission id="form" method="post"  action="getStockQuote.do"/>
</xf:model>

Interpretation 4
<html> (redundant, filtered out)
<head></head>
<body>
<br/>
</body>
</html>

<xf:group ref="stockquote"> (redundant, filtered out)
  <xf:input ref="symbol"><xf:label>Symbol</xf:label></xf:input>
  <xf:submit submission="form"><xf:label>Get Quote</xf:label></xf:submit>
</xf:group>
```

Appendix C

NVDL schema as a compound document

An NVDL schema is also a compound document as it may contain embedded subschema vocabularies inside `schema` elements or for example `xml:lang` attribute in messages. Moreover, an NVDL schema may be annotated using annotation languages. This implies, it makes sense to formalize such rules and create an NVDL script for NVDL. This section contains an example of such schema[1].

Example C.1. NVDL schema for NVDL

This NVDL schema allows any foreign vocabularies within any context. Only the context of the `schema` and `message` elements is handled differently. Known schema languages within `schema` element are send for validation and any unknown languages are simply allowed. Foreign vocabularies are rejected within `message`, but attributes from the XML's default namespace are sent for validation.

```
<rules startMode="root" ▶
xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
    <mode name="root">
        <namespace ns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0">
            <validate schema="nvdl.rng">
                <mode>
                    <anyNamespace>
                       <!-- allows any foreign elements and attributes ▶
within NVDL schemas -->
                        <allow useMode="attach"/>
                    </anyNamespace>
                </mode>
                <context path="schema">
                    <mode>
                      <!-- validate embedded Relax NG -->
                     <namespace ns="http://relaxng.org/ns/structure/1.0">
                        <validate ▶
```

[1] Another NVDL script for NVDL is also part of the [NVDL] specification.

```
schema="http://www.oasis-open.org/committees/relax-ng/relaxng.rng" ▶
useMode="attach"/>
                        </namespace>
                        <!-- validate embedded Schematron -->
                      <namespace ns="http://purl.oclc.org/dsdl/schematron">
                          <validate ▶
schema="http://www.schematron.com/iso-schematron.sch" useMode="attach"/>
                        </namespace>
                        <!-- put other known schema languages here for ▶
validation -->
                        <anyNamespace>
                          <allow useMode="attach"><!-- allows any unknown ▶
foreign schema languages without validation -->
                            </allow>
                        </anyNamespace>
                      </mode>
                  </context>
                  <context path="message">
                      <mode>
                          <namespace ▶
ns="http://www.w3.org/XML/1998/namespace" match="attributes">
                            <!-- default XML attributes are validated e. ▶
g. the xml:lang attribute -->
                            <validate schema="xmlattr.rng" ▶
useMode="attach"/>
                          </namespace>
                          <anyNamespace>
                              <reject/>
                          </anyNamespace>
                      </mode>
                  </context>
              </validate>
          </namespace>
      </mode>
      <mode name="attach">
        <anyNamespace><attach/></anyNamespace>
      </mode>
</rules>
```

Appendix D

Validation using triggers

Normally in NVDL, element sections are created based on their namespaces. Triggers allow to create element sections also based on the element's local name. The reason here is to allow NVDL-based validation of legacy SGML compound documents where different vocabularies were combined without the use of namespaces. The following diagram shows an example validation process. Even the input instance is a single-namespace document, using trigger it is decomposed into two sections which finally results into invoking two independent validation processes.

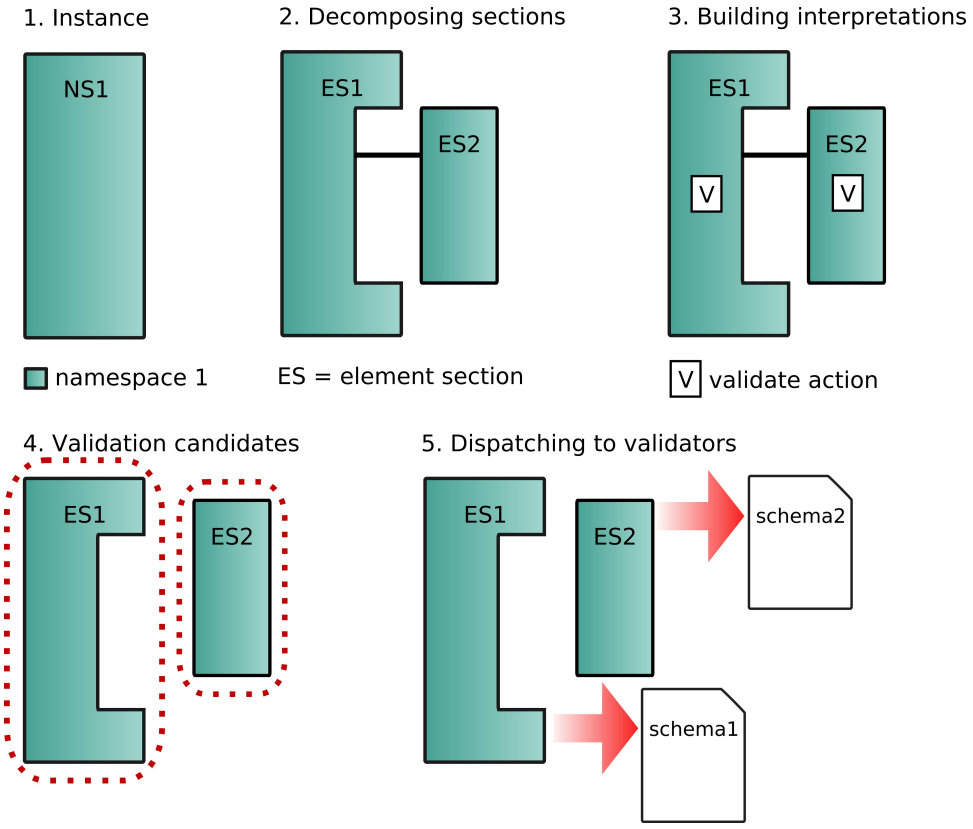

Figure D.1. Validation dispatching using triggers