RUHR
UNIVERSITÄT
BOCHUM

Feasibility and Performance Evaluation of Canonical XML

Student Research Project

Student:	Manuel Binna E-mail: manuel@binna.de Matriculation Number: 108004202162
Supervisor:	Dipl.-Inf. Meiko Jensen
Period:	20.07.2010 - 19.10.2010

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk
Faculty of Electrical Engineering and Information Technology
Ruhr University Bochum

Abstract

Within the boundaries of the XML specification, XML documents can be formatted in various ways without losing the logical equivalence of its content within the scope of the application. However, some applications like XML Signature cannot deal with this flexibility, thus needing a definite textual representation in order to distinguish changes which do or do not alter the logical equivalence of XML content. Canonical XML provides a method to transform textually different yet logically equivalent XML content into a single definite textual representation. This work evaluates the upcoming new major version Canonical XML Version 2.0 with respect to feasibility and performance.

Declaration

I hereby declare that the content of this thesis is a work of my own and that it is original to the best of my knowledge, except where indicated by references to other sources.

Location, Date	Signature

Table of Contents

1. Introduction

1.1. XML

Different XML documents can contain the same XML content, including permissible changes defined by the XML 1.0 and Namespaces in XML Recommendations. Given there are two XML documents, A’ and A”:

A’

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<note>
    <to>Attendees</to>
    <from>Manuel Binna</from>
    <heading>Seminar</heading>
    <body>Hello NDS!</body>
</note>
```

A”

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<note>
    <to>Attendees</to><from>Manuel Binna</from>
    <heading>Seminar</heading>
    <body>Hello NDS!</body>
</note>
```

Both documents, A’ and A”, are logically equivalent within the context of the application because they contain the same XML content. However, file A’ contains an additional newline character after the closing tag `</to>`. This character does not change the logical equivalence of both documents but the textual representation. This is a simple example to demonstrate the effect of a single modification without modifying the actual XML content. Other changes which also do not change the logical equivalence of an XML document are:

- Adding and removing whitespace characters between start-element tags.
- Adding whitespace characters between attributes in start-element tags.
- Modifying the order of namespaces and attributes in start-element tags.

XML processing applications like XML Signature [XMLSIG] for example require the input XML document to be in a well-defined format. XML Signature calculates a hash digest from the phyiscal representation of the content to be signed. A single change in the physical representation, i.e., bits, causes the digest method to output a completely different value. Because of the freedom to couch one and the same XML document in many different physical representations, XML processing applications between the signer and verifyer may modify the signed XML document in transit, hence invalidating the signature. Applications such as XML Signature therefore demand a method to create a single, definite physical representation from an arbitrary number of logically equivalent XML documents, the canonical form. This method is called canonicalization method.

1.2. Canonicalization

The goal of Canonical XML [C14N], [EXC-C14N], [C14N11], [C14N2] is to provide a mechanism for creating an XML document which is said to be in canonical form. The canonical form of an XML document is the single textual representation of one or more different XML documents which contain logically equivalent content within the scope of XML 1.0 [XML]. Canonical XML refers to XML content which is in canonical form. Canonicalization is the process to create canonical XML from arbitrary well-formed XML input.

Figure 1 illustrates the canonicalization of the documents A’ and A”. Both documents result in the output document A, which is in canonical form.

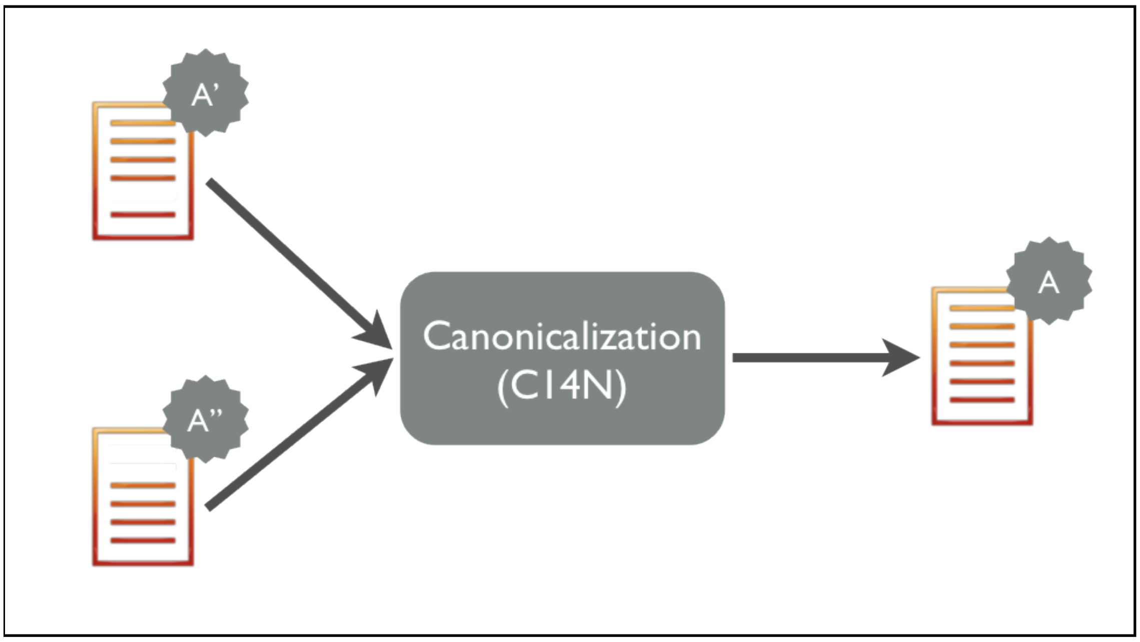


Figure 1: Canonicalization

C14N is an abbreveation of the term canonicalization. The W3C’s naming pattern to shorten very long words in their Recommendations takes the first and last letter of the word and substitutes the letters in between with the count of those letters. The abbreviation is mostly written in lowercase characters.

canonicalization ➝ c (anonicalizatio) n ➝ c (14 letters) c ➝ c14n

The W3C indicates the different versions of Canonical XML as follows:

Canonical XML Version 1.0	→	c14n
Exclusive Canonical XML 1.0	→	exc-c14n
Canonical XML Version 1.1	→	c14n11
Canonical XML Version 2.0	→	c14n2

1.3. History

This section gives a brief overview over the existing Recommendations of Canonical XML. It explains the major differences between them and highlights the reasons behind the release of a new version to complement or supersede an existing version.

1.3.1. Canonical XML Version 1.0

Canonical XML Version 1.0 was published on March 15, 2001 [C14N]. The Recommendation specifies two input parameters: one parameter is the input to be processed, the other specifies that the method should or should not process comments. The input is a well-formed XML document either as an octet stream or in the form of an XPath node-set. If the input is an octet stream, the *canonicalization method* first converts it into an XPath node-set. Attribute value normalization, entity reference resolution, and the inclusion of default attributes demand the usage of a validating XML processor. The method then processes the nodes of the node-set applying the rules defined by the Recommendation. The output octet stream is the *canonical form* of the input. It consists of the output characters for each node in ascending document order.

In detail, the *canonicalization method* defined in the Canonical XML Version 1.0 Recommendation performs the following tasks [C14N, section 1.1]:

- Remove the XML declaration and document type declaration (DTD)
- Encode the output in UTF-8
- Normalize line breaks to `#xA` before parsing
- Normalize attribute values
- Replace character and parsed entity references with their replacement content
- Replace CDATA sections with their character content
- Convert empty elements to pairs of start-tag and end-tag

- Normalize whitespace outside the document element and within start- and end-tags
- Retain whitespace in character content (excluding characters removed by line break normalization)
- Set attribute value delimiters to double quotes
- Replace special characters in attribute values and character content by character references
- Remove superfluous namespace declarations
- Add default attributes declared in the DTD
- Sort namespace declarations and attributes of each element by lexicographic order

1.3.2. Exclusive XML Canonicalization Version 1.0

Applications like XML Signature require a canonicalization method to be able to canonicalize a subdocument independent of the surrounding context yet yielding the same resulting octet stream. In those situations, mainly protocol applications, the signature must not break when the signed subdocument is removed from its original message and placed into a different message.

Exclusive XML Canonicalization, published on July 18, 2002, is a modification to the Canonical XML Version 1.0 Recommendation which fulfills that requirement. It adds an input parameter *inclusiveNamespacesPrefixList* to the other two parameters discussed in the previous section. It is mainly equivalent to Canonical XML Version 1.0, but introduces modifications for the processing of XML attributes and namespace declarations:

- The method does not inspect the ancestors of orphan nodes and copy XML attributes such as `xml:base`, `xml:space`, or `xml:lang` into the context.
- The *exclusive XML canonicalization method* processes the namespaces of the prefixes contained in the *inclusiveNamespacesPrefixList* such as Canonical XML Version 1.0 processes namespaces in general. The method outputs one of the other namespaces only when the current element node visibly utilizes it and when it was not already processed for an output ancestor.

An element node visibly utilizes a namespace declaration when itself or one of its attribute node descendants has a qualified name in which the prefix is the prefix of that namespace declaration. An element node visibly utilizes the default namespace when its qualified name is just the local name.

Example: A message sending application sends signed XML documents to another server. Consider the following (signed) input document for that application:

```
<msg:body xmlns:msg=”http://message”>
    Hello World!
</msg:body>
```

Listing 1

This message is signed and every modification which changes the textual representation of the message invalidates the signature. The message sending application puts the message in an enveloping element before processing it:

```
<app:header xmlns:app=”http://application”
    <msg:body xmlns:message=”http://message”>
        Hello World!
    </msg:body>
</app:header>
```

Listing 2

The receiver has to extract the original message and verify the signature. He applies the XPath expression

```
(//. | //@* | //namespace::*)[ancestor-or-self::msg:body]
```

to the received document and canonicalizes the node-set with the method described by Canonical XML Version 1.0. The resulting XML document is the following:

```
<msg:body xmlns:app=”http://application” xmlns:msg=”http://message”>
    Hello World!
</msg:body>
```

Listing 3

The canonicalization method includes the namespace declaration `xmlns:app=”http://application”` because `<msg:body>` is an orphan node. It imports the namespace declaration of the ancestor `<app:header>` into the context for the output of `<msg:body>`. The signature is broken, because the signed content is modified.

Exclusive XML Canonicalization outputs a namespace declaration only if the element node visibly utilizes it. The resulting XML document is the following:

```
<msg:body xmlns:msg=”http://message”>
    Hello World!
</msg:body>
```

Listing 4

This message is equivalent to the signed message and the receiver can successfully validate its signature.

1.3.3. Canonical XML Version 1.1

Canonical XML Version 1.1 was published on March 15, 2008, as a minor update to Canonical XML Version 1.0. The Recommendation adds rules to the processing of attributes in the XML namespace during the canonicalization of document subsets:

- Inheritance of the *simple inheritable attributes* `xml:space` and `xml:lang`
- No inheritance of `xml:id` attributes
- Fixup of `xml:base` attributes
- Processing of attributes in the XML namespace other than `xml:space`, `xml:lang`, `xml:id`, and `xml:base` as ordinary attributes

A *simple inheritable attribute* is an attribute that descendant element nodes of the current element node do not redeclare and thus inherit from their parent. If an element node redeclares a simple inheritable attribute, it does not inherit the attribute of its ancestor. The simple inheritable attributes specified by Canonical XML Version 1.1 are `xml:space` and `xml:lang`. The *canonicalization method* processes *simple inheritable attributes* as follows:

- Investigate the element nodes along the ancestor axis of the current element node and find the first occurrences of *simple inheritable attributes*.
- If the current element node redeclares *simple inheritable attributes* found in the previous step, those attributes are ignored.
- The remaining attributes found, but not redeclared, are added to the attribute axis of the current element node.

The Recommendation explicitly declares `xml:id` not a *simple inheritable attribute*. Thus, the canonicalization method does not process this attribute differently from normal attributes.

`xml:base` is not a simple inheritable attribute but requires specific treatment in contrast to a simple redeclaration. This processing is often called `xml:base` *fixup*. The method evaluates `xml:base` attribute values from omitted ancestor element nodes in the document subset and modifies the existing `xml:base` attribute value of the current element node with an updated value. The *canonicalization method* processes `xml:base` attributes as follows:

- Investigate the omitted element nodes along the ancestor axis of the current element node and store the values of `xml:base` attributes in an ordered list.
- Enumerate the ordered list in a way that the `xml:base` attribute values appear in reverse document order compared to their original appearance in the document subset.
- Use the *join-URI-References* function, declared in the Canonical XML Version 1.1 Recommendation, to calculate a new value from two consecutive values in the list.
- While enumerating the list, use the calculated value and the successive value in the list to calculate a new value with the *join-URI-References* function.
- Update the `xml:base` attribute value of the current element node by applying the *join-URI-References* function to the result of processing the list and the element node's value of the `xml:base` attribute.

The `xml:base` *fixup* is only performed if the document subset excludes an element node with an `xml:base` attribute. However, it is not performed if the document subset only excludes the `xml:attribute` node but not the element node.

Consider the following XML document (in analogy to the example found in [C14N11]):

```
<envelope xml:base="application/envelope">
  <header xml:base="hello">
    <title xml:base="world"></title>
  </header>
</envelope>
```

Listing 5

Assuming the document to be canonicalized does not contain the element `<header>`, the resulting output document would be:

```
<envelope xml:base="application/envelope">
  <title xml:base="hello/world"></title>
</envelope>
```

Listing 6

1.3.4. Canonical XML Version 2.0

At this moment, Canonical XML Version 2.0 has not been published as W3C Recommendation yet. This thesis evaluates the Canonical XML Version 2.0 Editor's Draft released on June 22, 2010.

Canonical XML Version 2.0 is a new major version of the Recommendation and merges the functionality of Canonical XML Version 1.0, Exclusive XML Canonicalization Version 1.0, and Canonical XML Version 1.1 into one algorithm configurable through parameters. The next section lists the different parameters and explains their meaning.

1.3.4.1. Parameters

Canonical XML Version 2.0 introduces several parameters which influence the output of the canonicalization method:

- **exclusiveMode** (value: true | false)
  Process namespace declarations in either inclusive mode (as described in [C14N] and [C14N11]) or exclusive mode (as described in [EXC-C14N]). The default value is "false".

- **inclusiveNamespacePrefixList** (value: list of prefixes)
  A list of namespace prefixes which should be treated inclusively when using exclusive mode. The namespace prefixes are separated by a space. This parameter has no effect in inclusive mode. The default value is an empty list.

- **ignoreComments** (value: true | false)
  Omit or comprise comments during processing. If this parameter value is "true", comments in the input are not processed. The default value is "true".

- **trimTextNodes** (value: true | false)
  With this parameter set to "true", leading and trailing whitespace characters are removed from text content. When the parameter value is "false" or an ancestor of the text content contains the attribute `xml:space="preserve"`, the canonicalization method does not trim the text. The default value is "false".

- **serialization** (value: serializeXML | serializeEXI)
  Depending on this parameter value, the output format is either XML or EXI. EXI stands for Efficient XML Interchange [EXI]. Its goal is "to develop a specification for an encoding format that allows efficient interchange of the XML Information Set and to illustrate effective processor implementations of

that encoding". This thesis omits the evaluation of EXI serialization. The default value is "serializeXML".

- **prefixRewrite** (value: none | sequential | derived)
  This parameter defines how namespace prefixes are handled. If the value is "none", prefixes are not modified. If the value is "sequential", prefixes are rewritten to nX, where X denotes an integer value. The *canonicalization method* increments the value of a counter each time it processes a new namespace declaration. The rewritten prefix nX contains the counter's current value at that time. If the parameter value is "derived", prefixes are rewritten to nD, where D is the SHA1 hash value of the corresponding namespace URI. The default value is "none".

- **sortAttributes** (value: true | false)
  Attributes, including namespace declarations, are sorted by ascending (lexicographic) order, if this parameter is set to "true". Otherwise, attributes are not sorted and remain in the same order in which they appear in the input. The default value is "true".

- **xmlAncestors** (value: inherit | none)
  If this parameter is set to "none", the values of XML attributes (`xml:base`, `xml:lang`, and `xml:space`) of ancestors are ignored when processing the current element. If the parameter value is "inherit", the *simple inheritable attributes* of ancestors, `xml:space` and `xml:lang`, are inherited. The *canonicalization method* also performs the `xml:base` *fixup* to combine ancestor values according to [XMLBASE]. The default value is "inherit".

- **xsiTypeAware** (value: true | false)
  When this parameter is set to "true", the value of `xsi:type` attributes are evaluated for the usage of namespace prefixes. If set to "false", `xsi:type` attributes are not treated separately from the other attributes. The default value is "false". This work omits the evaluation of this parameter since upcoming versions of the Editor's Draft will use other mechanisms to indicate the processing of namespace prefixes in content.

To support the former Recommendations, the *canonicalization method* defined by Canonical XML Version 2.0 can be parameterized appropriately:

Canonical XML Version 1.0	
exclusiveMode	false
inclusiveNamespacePrefixList	<empty list>
ignoreComments	true / false
trimTextNodes	false
serialization	serializeXML
prefixRewrite	none
sortAttributes	true
xmlAncestors	none
xsiTypeAware	<empty list>

Figure 2

Exclusive XML Canonicalization Version 1.0	
exclusiveMode	true
inclusiveNamespacePrefixList	<empty list>
ignoreComments	true / false
trimTextNodes	false
serialization	serializeXML
prefixRewrite	none
sortAttributes	true
xmlAncestors	none
xsiTypeAware	<empty list>

Figure 3

Canonical XML Version 1.1	
exclusiveMode	false
inclusiveNamespacePrefixList	<empty list>
ignoreComments	true / false
trimTextNodes	false
serialization	serializeXML
prefixRewrite	none
sortAttributes	true
xmlAncestors	inherit
xsiTypeAware	<empty list>

Figure 4

1.3.4.1. Improvements

Canonical XML Version 2.0 aims to improves many issues of older versions:

- **Performance**: Canonical XML Version 2.0 restricts the input of the *canonicalization method* so that the implementation can always perform a simple tree walk by visiting each node exactly once. The algorithm can perform better because it is not based on the relatively complex XPath node-set. It processes only the nodes which are canonicalized and visits each node only once.

- **Streaming**: Implementations based on a streaming XML parser are easier to implement because a simple tree walk is sufficient to process the input.

- **Robustness**: Canonical XML Version 2.0 enhances the robustness against e.g. signature breakage through the (optional) removal of leading and trailing whitespace in text content (trimTextNodes), prefix rewriting (prefixRewrite), and the processing of qualified names in content (xsiTypeAware). This decreases the impact of pretty-printing, i.e. the insertion of whitespace characters to make the document more readable.

- **Simplicity**: Because Canonical XML Version 1.0, Exclusive XML Canonicalization Version 1.0, and Canonical XML Version 1.1 use an XPath node-set, implementations need to use an XPath library. By not using an XPath node-set, implementations of Canonical XML Version 2.0 can always perform a simple tree walk to process the input. This enables them to be faster, simpler and less error-prone.

1.4. Canonicalization and XML Signature

Canonical XML Version 2.0 can be used as a canonicalization algorithm for XML Signature. The identifier is:

```
http://www.w3.org/2010/xml-c14n2
```

The following example shows the usage for XML Signature. Every parameter is optional and has a default value.

```
...

<CanonicalizationMethod Algorithm=”http://www.w3.org/2010/xml-
c14n2”>
  <ExclusiveMode>true</ExclusiveMode>
  <InclusiveNamespaces>
    <PrefixList></PrefixList>
  </InclusiveNamespaces>
  <IgnoreComments>true</IgnoreComments>
  <TrimTextNodes>false</TrimTextNodes>
  <Serialization>XML</Serialization>
  <PrefixRewrite>none</PrefixRewrite>
  <SortAttributes>true</SortAttributes>
  <XmlAncestors>none</XmlAncestors>
  <XsiTypeAware>false</XsiTypeAware>
</CanonicalizationMethod>

...
```

Listing 7

Figure 5 illustrates the process of signing an XML document:

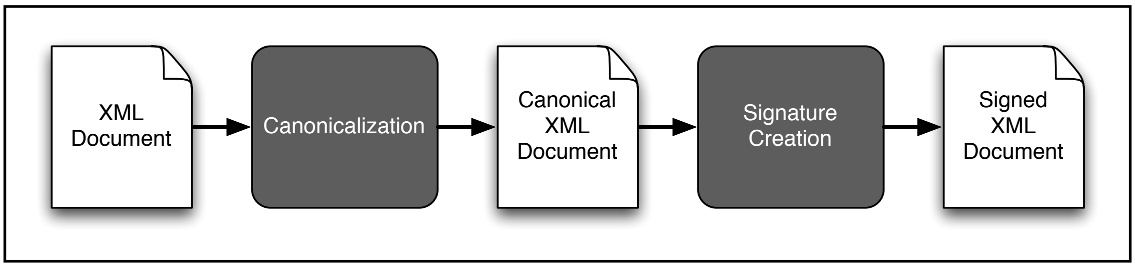


Figure 5: Signing an XML Document

Figure 6 shows the verification of a signed XML document:

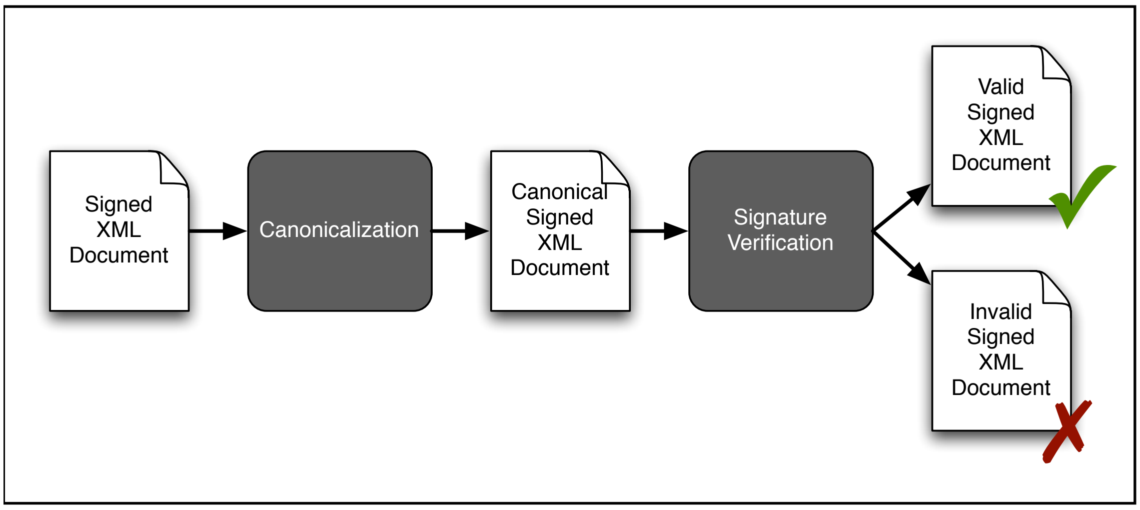

Signed XML Document
Canonicalization
Canonical Signed XML Document
Signature Verification
Valid Signed XML Document
Invalid Signed XML Document

Figure 6: Verifying a Signed XML Document

2. Feasibility Evaluation

The main part of the feasibility evaluation is the implementation of the Canonical XML Version 2.0 Editor's Draft of June 22, 2010 [C14N2]. This chapter discusses the architecture of the developed software and the experience gained during the implementation.

2.1. Programming Language

The programming languages Java, C / Objective-C, and Ruby were considered at the beginning of this project to be used for the implementation of the Canonical XML Version 2.0 Editor's Draft.

Java has powerful XML capabilities and is popular in academic world these days. Among computer science students Java is a very common language. It is well-known and understood by many people involved in the field of XML processing.

Objective-C is a superset of C which adds object-oriented capabilities to the C programming language. It also allows programmers to embed source code written in C. XML processing in C is efficient and straightforward given that a library like libxml2 [LIBXML2] is used. Because XML processing heavily involves the manipulation of strings, the programmer has to take care when using C-based languages (C, Objective-C and C++). Manipulating character pointers and dealing with the string termination character '\0' can be frustrating for the programmer and can lead to bugs and vulnerabilities.

Ruby is a full object-oriented programming language. It was first developed by Yukihiro Matsumoto in 1995. The current version of Ruby is 1.9.2. It is available under an open source license. Ruby is dynamically typed. Objects do not have a specific type like objects in statically typed programming languages like Java. An object merely behaves like something. This approach is called *Duck Typing*. When a message is sent to an object the corresponding method of that object is invoked. Everything is an object in Ruby: numbers, strings, classes, object instances, etc.. Because Ruby is an interpreted language, i.e. Ruby programs are scripts interpreted by an interpreter, its performance is lacking behind implementations written, e.g., in C, where the source code is compiled into assembly code, or Java, where the source code is compiled to byte code.

Different factors led to the decision to choose Ruby as the programming language for the implementation:

- Ruby is available on many operating systems, including Mac OS X, Windows, and Linux.
- Ruby provides an extensive collection of APIs like ordered lists, key-value stores, and objects for string manipulation. In fact, as a developer, string manipulation in Ruby is very similar to string manipulation in Java.
- Because Java is often used as the programming language for the implementation of XML processing Recommendations, this thesis takes the refreshing approach of not using Java but Ruby.

2.1.1. Ruby Bindings for libxml2: libxml-ruby

Ruby bindings of libxml2, the widely used open source XML library from the GNOME project, exist under the name *libxml-ruby*. The bindings provide Ruby classes to use the underlying C-APIs of libxml2. Those classes are implemented with the *Ruby C Language API* which enables the implementation of Ruby classes in the C programming language. Therefore, *libxml-ruby* can leverage the maturity of libxml2 and provide a clean object-oriented interface to Ruby programmers.

The implementation of this work uses the current version of libxml-ruby (v1.1.3) and the current version of libxml2 (v2.7.7).

2.2. Parsers

Before discussing the details of the implementation, the three different kinds of XML parsers available in libxml-ruby are illustrated. Each one works different from the others in essential ways.

2.2.1. Tree-based parsing (DOM)

Parsers which use the principle of tree-based parsing process an XML document as one chunk and assemble a tree of objects corresponding to the content of the XML document in memory. This tree of objects is called the *Document Object Model (DOM)*. The contained objects are called *nodes*. Processing an XML document with the tree-based approach enables random access to the nodes through the *DOM*.

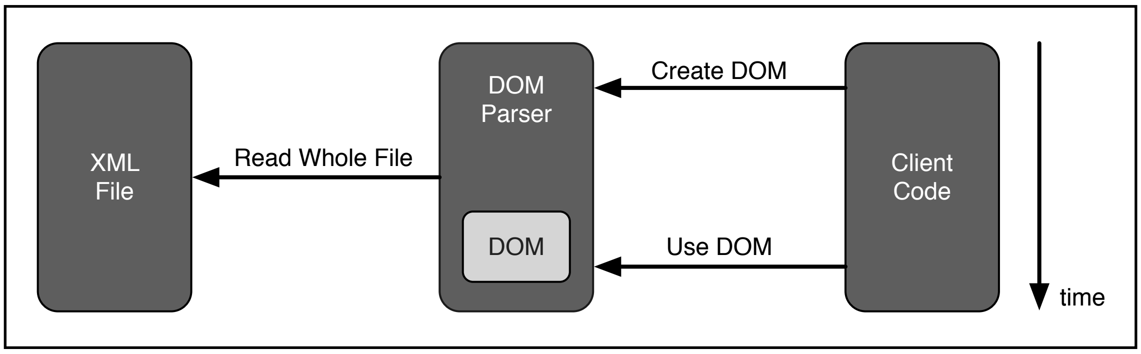


Figure 7: DOM Parser

The main advantage of having the full representation of an XML document in memory is the simple and flexible access to the contained nodes. The approach enables accessing nodes randomly and multiple times. However, when processing very large XML documents the tree-based approach might exceed the available resources, i.e. space in the main memory.

libxml-ruby provides a tree-based parser via the class `LibXML::XML::Parser`.

2.2.2. Push Parsing (SAX)

The *Simple API for XML (SAX)* describes a push parser which reads an XML document one chunk at a time generating events like “DidStartElement”, “FoundCharacters”, or “DidEndElement” for the different artifacts of the chunk. The approach is also called event-based parsing. The client of the parser registers callback functions which are invoked when the parser encounters corresponding events. The client can access the data via the parameters provided to the callback (see Figure 8). After an event is processed through the invocation of a callback function in the client code the parser continues with the next event. The parser does not hold state between the invocations of the callback functions. This task has to be performed by the client code.

The process is called *push parsing* because the parser pushes the parsed data to the client code by invoking callback functions and providing access to the data through the parameters of that functions.

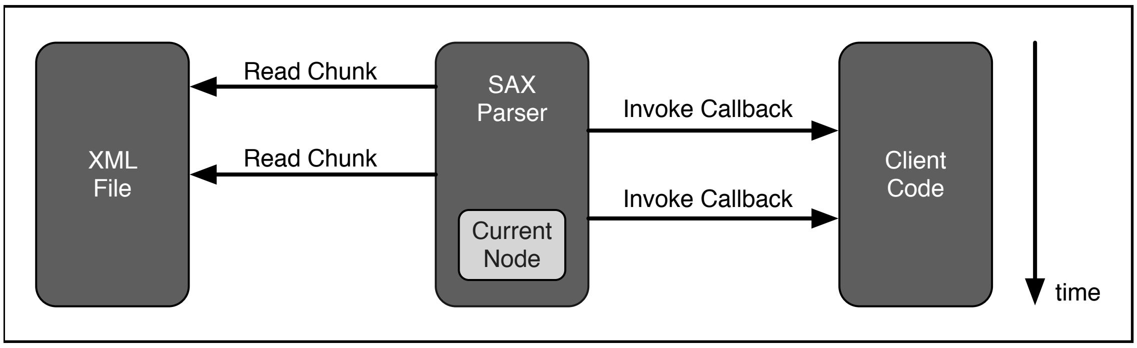


Figure 8: SAX Parser

When it comes to memory usage, *push parsing* has an advantage over the tree-based approach: a push parser processes the input in small chunks. Only a small portion of an input file or input stream is read and parsed at a single time. This allows the parser to operate with a constant tiny memory footprint regardless of the size of the input.

The disadvantages of SAX are that "the programming model is relatively complex, [it is] not well standardized, cannot provide validation directly, makes entity, namespace and base processing relatively hard" [TEXTREADER].

libxml-ruby provides a push parser via the class `LibXML::XML::SaxParser`.

2.2.3. Pull Parsing (StAX)

The *Streaming API for XML (StAX)* provides a *pull parsing* mechanism. The *StAX* parser maintains a cursor to the current node in the XML content. The client code repeatedly moves the cursor to the next node in document order and pulls the information from that node. The client is responsible for the logic to traverse the XML document. The parser just moves the cursor to the next node when told by the client code.

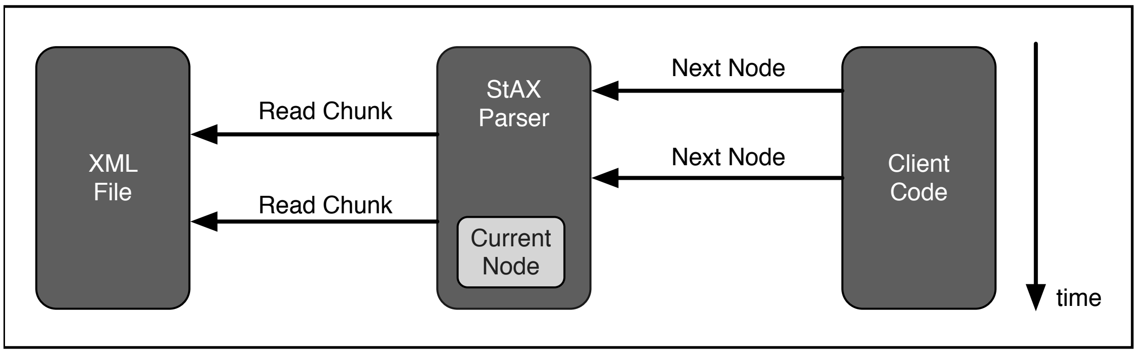


Figure 9: StAX Parser

The advantage of using *StAX* over *SAX* is its simpler and cleaner API. Like *SAX* it processes the input in small chunks providing a small and constant memory footprint regardless of the size of the input.

libxml-ruby provides a pull parser via the class `LibXML::XML::Reader`. It "provides a simpler, alternative way of parsing an XML document [...] [It] acts like a cursor going forward in a document stream, stopping at each node it encounters." [LIBXML-RUBY-API].

2.3. Validating vs. Non-validating Parsers

`LibXML::XML::Reader` is a validating parser. `LibXML::XML::SaxParser` is a non-validating parser. The Canonical XML Recommendations require an implementation to include default attributes and to resolve entity references. A validating parser provides the support to process the information in a DTD when parsing an XML document. It can then incorporate that data when parsing the actual XML document. Adding default attributes or replacing parsed entities with their replacement content during canonicalization is easy when using a validating parser.

The non-validating parser `LibXML::XML::SaxParser` does not automatically include information from a DTD. It invokes registered callback functions for the events related to DTD, but the developer has to extract the necessary DTD information and incorporate it into the canonicalized output. `LibXML::XML::SaxParser` invokes the following methods related to DTD in the client code [LIBXML-RUBY-API]:

- `on_external_subset(name, external_id, system_id)`
- `on_has_external_subset()`
- `on_has_internal_subset()`
- `on_internal_subset(name, external_id, system_id)`
- `on_reference(name)`
- `on_is_standalone()`

2.4. Implementation of Canonical XML Version 2.0

Two different implementations were implemented for this thesis. The one implementation is based on the StAX parser of libxml-ruby, the other implementation is based on the SAX parser of libxml-ruby. Since the additional effort to implement the processing of DTD information with the SAX parser of libxml-ruby is huge, it is omitted from the implementation. The SAX-based implementation is therefore not able to add default attributes to elements and cannot replace parsed entity references with their replacement content. The StAX-based implementation aims to provide the full functionality of Canonical XML Version 2.0.

2.4.1. Assumptions About the Input Model

The input of Canonical XML Version 2.0 consists of an inclusion list, an exclusion list, and a set of parameters. The inclusion list specifies the elements which should be canonicalized. The list can either contain the whole input document or a list of elements. The exclusion list specifies the elements and attributes which should not be canonicalized. It is composed of elements and attributes. During canonicalization only the elements and descendants provided in the inclusion list are canonicalized. If the exclusion list contains a particular element (or attribute), that element (or attribute) and all descendants of that element are omitted during canonicalization.

The selection of elements to be canonicalized in a streaming implementation of Canonical XML Version 2.0 is done with the XML Signature Streaming Profile of XPath 1.0 [STREAM_XPATH]. The mechanism in a DOM-based implementation is a “very limited form of the generic XPath node-set” ([C14N2] 2.1 Data Model). The implementation of the selection algorithm is not part of this thesis and was not implemented.

The selection can lead to situations where the input of the canonicalization method is not well-formed and the parser cannot parse the input. The following example demonstrates this situation:

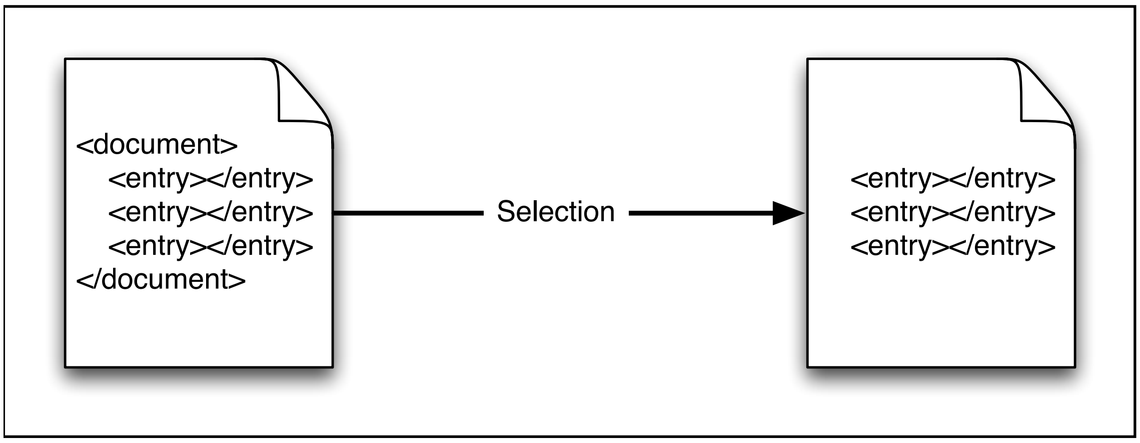
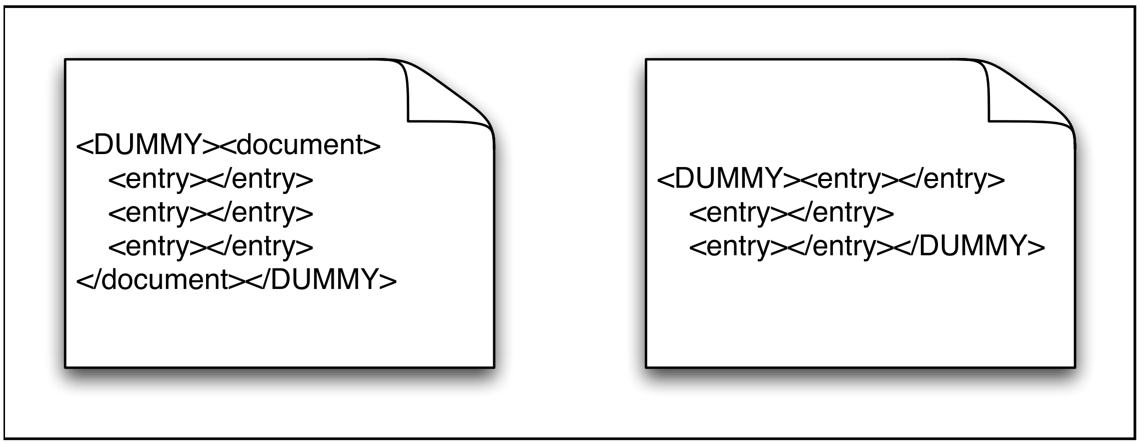


Figure 10: Selection Mechanism

After performing the selection, the resulting document is not well-formed. `LibXML::XML::SaxParser` and `LibXML::XML::Reader` require the input document to be well-formed. Prior to selection the input is always a well-formed document. Our streaming implementation parses the whole input document and performs the selection during the parsing. The architecture anticipates the addition of the selection algorithms at a later date.

In order to compensate for situations in which the resulting input document would not be well-formed, the input document uses a dummy element to envelop the actual input (see Figure 11).



```
<DUMMY><document>
   <entry></entry>
   <entry></entry>
   <entry></entry>
</document></DUMMY>
```

```
<DUMMY><entry></entry>
   <entry></entry>
   <entry></entry></DUMMY>
```

Figure 11: Dummy Element Envelops The Actual Input

2.4.2. SAX-based Architecture

The SAX-based architecture uses the class `LibXML::XML::SaxParser` to parse the input XML document.

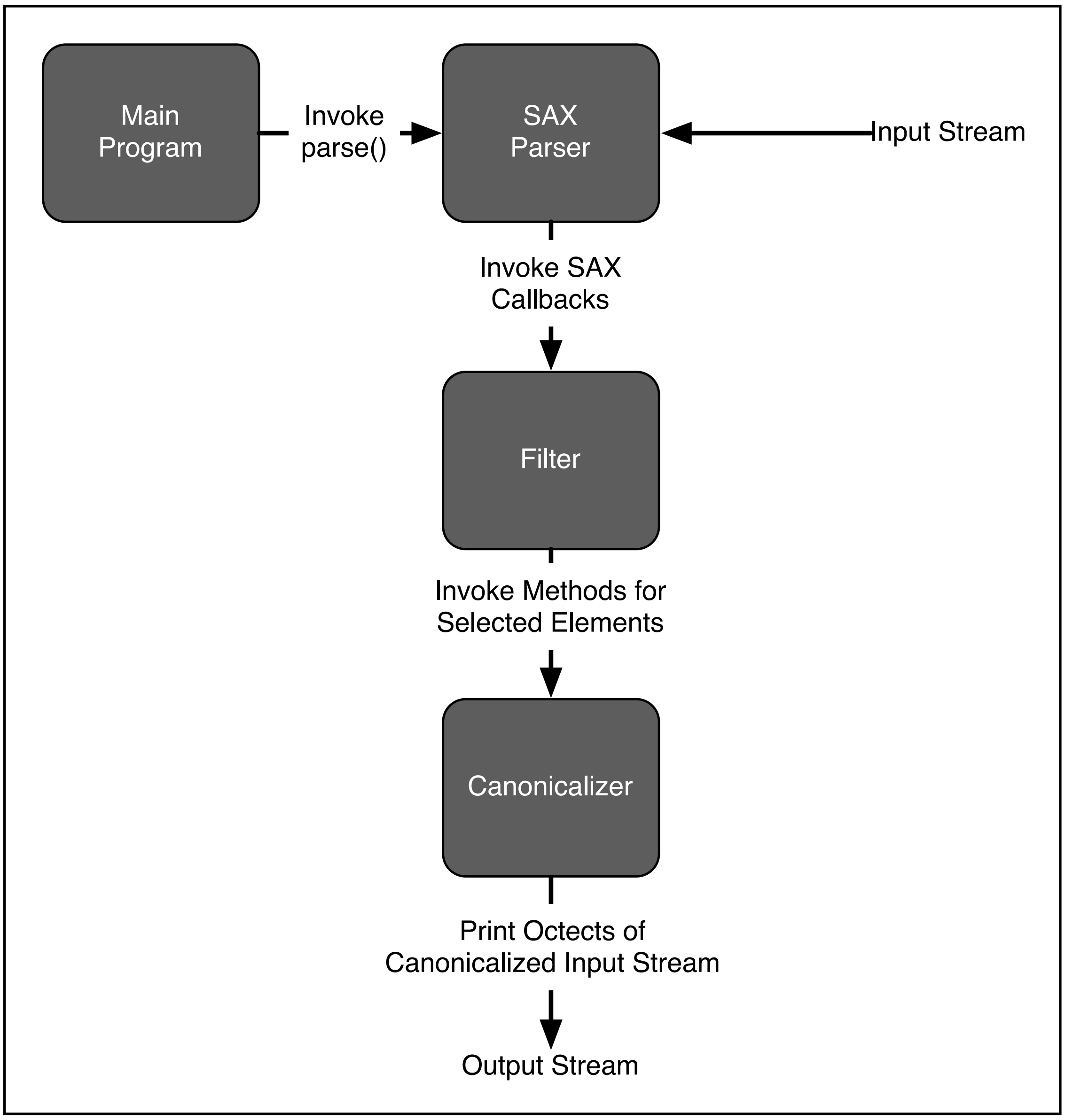


Figure 12: Architecture of the SAX-based Implementation

- **Main Program:** The main program parses the command line parameters, creates and configures the involved objects, and starts the canonicalization process.

- **SAX Parser:** The SAX parser is an instance of `LibXML::XML::SaxParser`. It invokes the callbacks implemented by the Filter.

- **Input Stream:** The well-formed XML input document to be canonicalized is read through an input stream while processing it chunk-wise.

- **Filter:** The Filter selects elements of the input stream which are to be canonicalized. It also selects elements and attributes which should not to be canonicalized. It uses the XML Signature Streaming Profile of XPath 1.0 [STREAM_XPATH] to perform its work.
  For every start-element event the Filter receives it checks if the XPath expression for the included elements matches the element. If it does not match, the start-element event is ignored. If it does match, it checks if the XPath expression for the excluded elements matches the element. If it does match, the start-element event is ignored. Otherwise the element has to be canonicalized. Before propagating the event to the Canonicalizer, its attributes are checked if the XPath expression for excluded elements and attributes matches them. If the expression matches it is removed from the list of attributes. The Filter is currently implemented as a dummy without the described selection logic.

- **Canonicalizer:** The Filter invokes the relevant methods on the Canonicalizer to process the relevant events and outputs the selected elements canonicalized as an octet stream.

- **BaseURI (not shown in Figure 12)**: The class BaseURI implements the modified *join-URI-References* function according to the Canonical XML Version 2.0 Editor's Draft. It is invoked when the Canonicalizer needs to calculate a new value for `xml:base`.

2.4.2.1. Canonicalizer

The operating mode of the SAX-based implementation is shown in Figure 13. The implementation of the Canonicalizer uses principles described in the section "Pseudocode" of the Canonical XML Version 2.0 Editor's Draft.

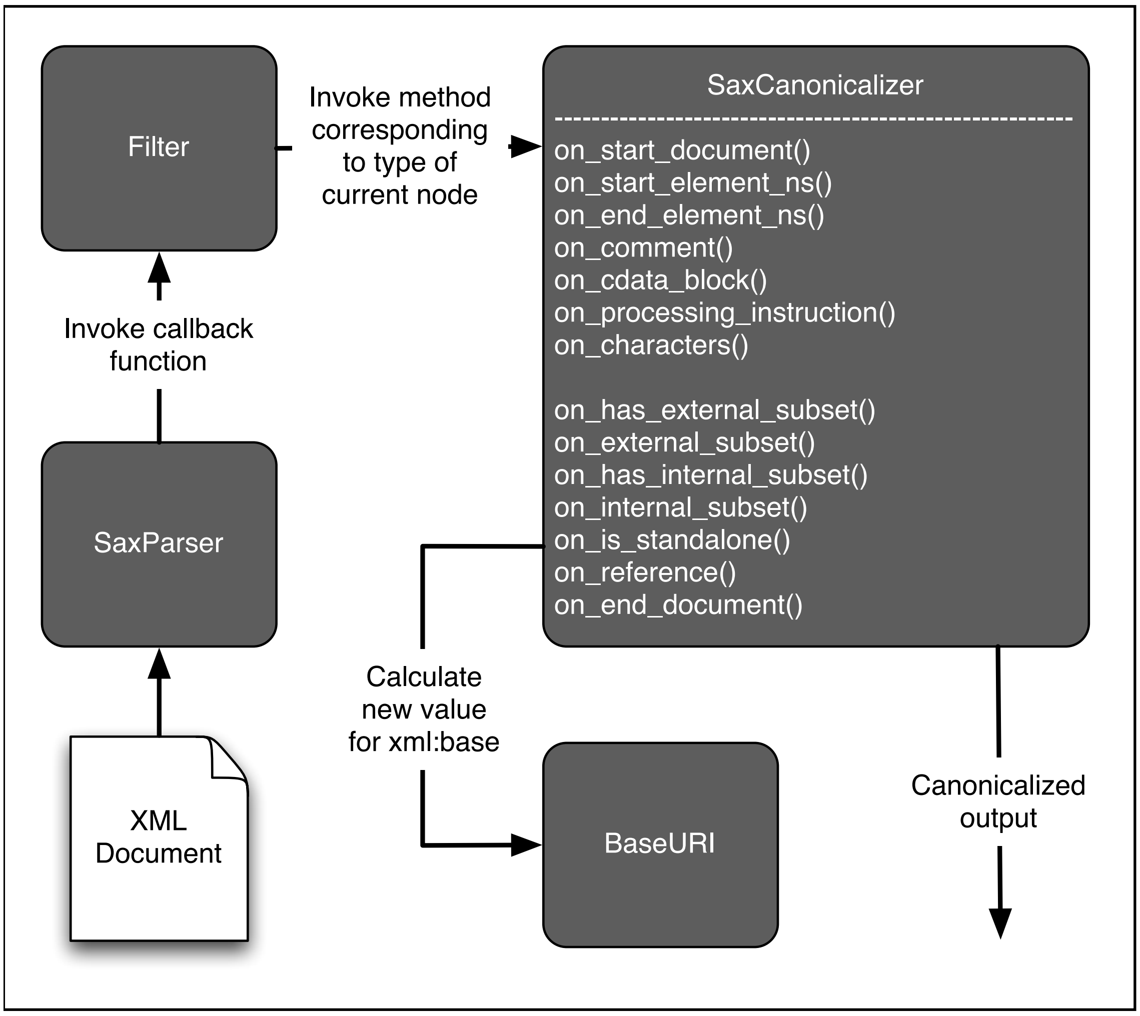


Figure 13: Operating Mode of the SAX-based Implementation

The methods of SaxCanonicalizer implement the logic to canonicalize the elements selected by the Filter.

`on_start_document()`
- Initialize the state of the Canonicalizer.
- Add the empty prefix with an empty URI to the namespace context.

`on_start_element_ns(name, attributes, prefix, uri, namespaces)`
- Call the Proc for processing and outputting found characters (see below). A Proc implements a Closure in Ruby.
- Declare a Proc to check if the element visibly utilizes a given prefix. Store the Proc in an instance variable to be used in succeeding methods. The Proc has access to the surrounding scope and can access the method parameters even when called from another context.
- Declare a Proc for processing and outputting the characters found and store it in an instance variable.

- Invoke the method `process_namespaces(namespaces)` and store the returned Array in a local variable.
- Invoke the method `process_attribues(attributes)` which processes the attributes in-place.
- Assemble and output the complete start tag of the current element including the qualified name, namespace, and attributes.

`on_end_element_ns(name, prefix, uri)`
- Call the Proc for processing and outputting found characters.
- Assemble and output the end-tag of the current element.
- Invoke the method `remove_namespaces_from_context()` to clean up the namespace context.
- Invoke the method `remove_xml_attributes_from_context()` to clean up the xml attribute context.

`on_comment(msg)`
- Call the Proc for processing and outputting found characters.
- Return if parameter `ignoreComments="true"`.
- Otherwhise assemble and output the comment.

`on_cdata_block(cdata)`
- Call the Proc for processing and outputting found characters.
- Replace undefined characters with their numeric character references. Convert `&`, `<`, and `>` to `&amp;`, `&lt;`, and `&gt;`, respectively. Output the result.

`on_processing_instruction(target, data)`
- Call the Proc for processing and outputting found characters.
- Assembles and outputs the processing instruction.

`on_characters(chars)`
- Append the String `chars` to the instance variable which stores the found characters.

The methods mentioned above use internal methods to split up the work involved:

`process_namespaces(namespaces)`
- Invoke `add_namespaces_to_context(namespaces)`.
- Add the namespaces to be output from the namespace context to an Array.
- Optionally rewrite the namespace prefixes in the Array.
- Optionally sort the Array by lexicographic order of namespace URI.
- Return the Array of namespaces to be output.

`add_namespaces_to_context(namespaces)`

- Add the provided namespaces to the namespace context.
- Before adding the namespace replace undefined characters in the namespace URI with their numeric character references, convert `&`, `<`, `>`, and " to `&amp;`, `&lt;`, `&gt;`, and `&quot;`. Quote the result using double quotes.

`remove_namespaces_from_context`

- Remove the deprecated namespaces from the namespace context, i.e. namespaces of element which occured at a higher depth in the document than the current depth of the processing.

`process_attributes(attributes)`

- Optionally rewrite the attribute prefixes.
- Replace special characters in the attribute values.
- Optionally store xml attributes in the xml attribute context (`xmlAncestors="true"`), performing the modified *join-URI-References* function for xml:base defined by the Canonical XML Version 2.0 Editor's Draft.
- Optionally add xml attributes from the xml attribute context to the attributes Array (`xmlAncestors="true"`).
- Optionally sort the attributes Array (`sortAttributes="true"`).

`remove_xml_attributes_from_context()`

- Remove the deprecated xml attributes from the namespace context, i.e. xml attributes of elements which occured deeper in the document than the current depth of the processing.

2.4.2.2. Modifications on libxml-ruby

During the work on this thesis it was found that `LibXML::XML::SaxParser` does not provide namespace prefixes of attributes to its callback handler (a class which includes `LibXML::XML::SaxParser::Callbacks`) when invoking the callback method `on_start_element_ns()`. libxml2 provides namespace prefixes of attributes. `LibXML::XML::SaxParser::Callbacks` is implemented in the file `ruby_xml_sax2_handler.c`. Listing 8 shows the important part of `start_element_ns_callback()` which extracts the information provided by libxml2's SAX2 parser and constructs Ruby objects populated with it. The function provides only the attribute name and value but not the namespace prefix and the corresponding namespace URI. The original code was stripped and slightly re-formatted to better fit into this document.

```
static void
start_element_ns_callback(
  void *ctx,
  const xmlChar *xlocalname,
  const xmlChar *xprefix,
  const xmlChar *xURI,
  int nb_namespaces,
  const xmlChar **xnamespaces,
  int nb_attributes,
  int nb_defaulted,
  const xmlChar **xattributes)
{

  ...

  int i;
  for (i = 0; i < nb_attributes * 5; i+=5)
  {
    VALUE attrName = rb_str_new2(xattributes[i+0]);
    VALUE attrValue =
    rb_str_new(xattributes[i+3], xattributes[i+4] - xattributes[i+3]);
    /* VALUE attrPrefix =
           xattributes[i+1] ? rb_str_new2(xattributes[i+1]) : Qnil;
       VALUE attrURI =
           xattributes[i+2] ? rb_str_new2(xattributes[i+2]) : Qnil;
     */

  rb_hash_aset(attributes, attrName, attrValue);

  ...

}
```

Listing 8: `start_element_ns_callback()`

Since the *canonicalization method* relies on the complete information of an attribute (prefix, localname, value), `start_element_ns_callback()` had to be modified to provide the available information. Appendix A discusses the modification of libxml-ruby in greater detail.

2.4.2.3. Processing Character Content

The SAX parser may invoke the callback function `on_characters()` multiple times while parsing the character content. It invokes the callback function multiple times especially when the character content contains entity references. This behaviour complicates the trimming of content because it must only occur at the beginning and at the end of the whole content. Therefore the logic to handle the trimming cannot be implemented in the callback `on_characters()`.

Our implementation uses a Proc [RUBY-PROC] in the method `on_start_element_ns()` to store the logic for processing the character content and capture the current context. The Proc is stored as an instance variable of the Canonicalizer. The implementation of `on_characters()` appends the characters found to a String stored as an instance variable. When the next event occurs and the SAX parser invokes a callback other than `on_characters ()`, the implementation of the callback checks if there are any characters to be output. If there are characters available, it calls the Proc. The Proc accesses the character content previously assembled into the String instance variable and performs the trimming with the method `String#strip!`.

The Ruby class `String` is implemented with the *Ruby C Language API*. It "holds and manipulates an arbitrary sequence of bytes, typically representing characters" [RUBY-RDOC]. The method `String#strip!` internally invokes the function `rb_str_lstrip_bang()` to strip the string at the left hand side and `rb_str_rstrip_bang()` to strip the string at the right hand side. Both functions use pointer arithmetic to iterate over the codepoints of the string to detect if codepoints have to be stripped. The iteration stops if the codepoint evaluated is no whitespace. `rb_str_lstrip_bang()` uses `memmove()` to copy the bytes of the string so that it begins with a non-whitespace codepoint. `rb_str_rstrip_bang()` works similiarly for the right hand side of the string.

2.4.2.4. Sorting

Namespaces and attributes are sorted with the method `Array#sort!` which uses Quicksort and is highly optimized for Ruby objects [RUBY-SORT]. `Array` is implemented with the *Ruby C Language API*.

2.4.3. StAX-based Architecture

The StAX-based architecture uses the class `LibXML::XML::Reader` to parse the input document.

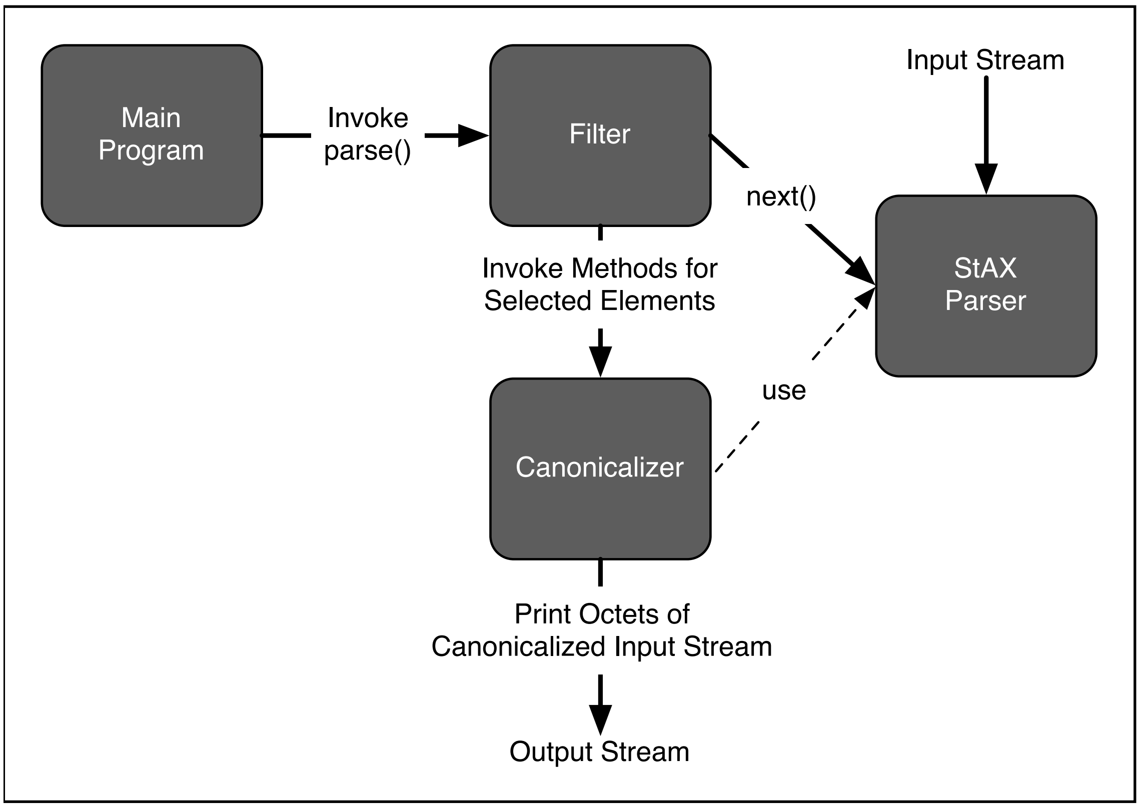


Figure 14: Architecture of the StAX-based Implementation

- **Main Program:** The main program parses the command line parameters, creates and configures the involved objects, and starts the canonicalization process.

- **StAX Parser:** The StAX parser is an instance of `LibXML::XML::Reader`. It acts like a cursor to iterate over the XML content being parsed.

- **Input Stream:** The well-formed XML input document to be canonicalized is read through an input stream processing it chunk-wise.

- **Filter:** The Filter iterates over the XML content by incrementing the cursor of the StAX parser one node at a time. It selects the nodes to be canonicalized with the same mechanism as the Filter of the SAX-based architecture. The filter uses the XML Signature Streaming Profile of XPath 1.0 to perform its work. If a node is selected to be canonicalized, the Filter invokes the corresponding method on the Canonicalizer.
  The Filter is currently implemented as a dummy without the desribed

selection logic.

- **Canonicalizer:** The Canonicalizer processes the selected nodes and outputs the canonicalized XML document. It uses the state of the StAX parser to obtain the data to be canonicalized.

- **BaseURI (not shown in Figure 14)**: The class BaseURI implements the modified join-URI-References function according to the Canonical XML Version 2.0 Editor's Draft. It is invoked when the Canonicalizer needs to calculate a new value for `xml:base`.

2.4.3.1. Canonicalizer

The operating mode of the StAX-based implementation is shown in Figure 15. Like the SAX-based implementation, the Canonicalizer uses principles described in the section "Pseudocode" of the Canonical XML Version 2.0 Editor's Draft.

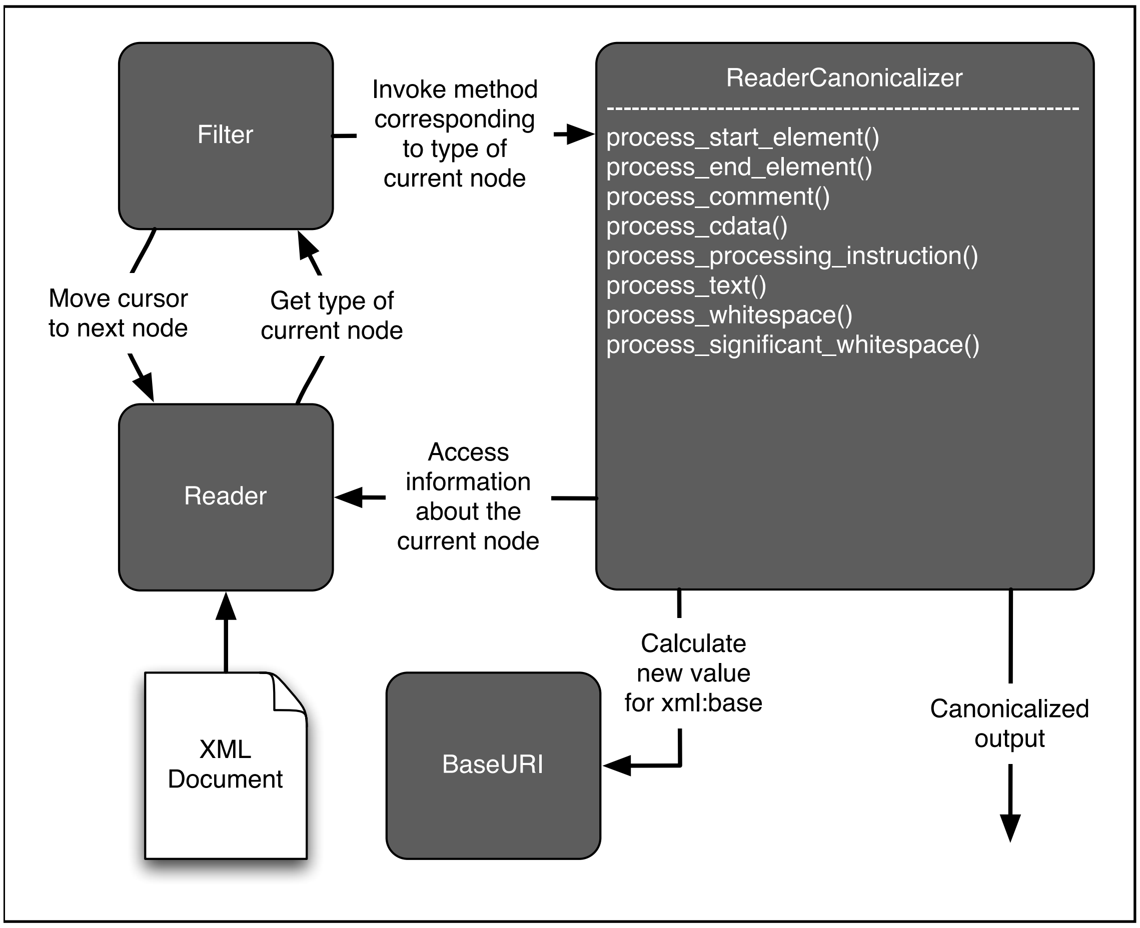


Figure 15: Operating Mode of the StAX-based Implementation

The methods of ReaderCanonicalizer implement the logic to canonicalize the elements selected by the Filter.

`process_start_element()`
- Push the previous value of `xml:base` on the base URI stack.
- Deep copy the previous value of `xml:base`.
- Invoke the method `process_namespaces()` and store the returned Array in a local variable.
- Assemble and output the QName and namespace declarations.
- Invoke the method `process_attribues()` which processes the attributes to be output and returns them in an Array.
- Output the attributes found in the Array.

`process_end_element()`
- Output the end tag of the element.
- Invoke `remove_namespaces()` to remove deprecated namespaces from the namespace context.
- Restore the previous value of xml:base

`process_comment()`
- Assemble and output the comment if `ignoreComments=”false”`.

`process_cdata()`
- Replace all `&`, <, and > by `&amp;`, `&lt;`, and `&gt;`, respectively.
- Output the result.

`process_processing_instruction()`
- Assemble and output the processing instruction.

`process_text()`
- Replace all `&`, <, and > by `&amp;`, `&lt;`, and `&gt;`, respectively.
- Replace all occurences of the character `#xD` with the character reference `&#xD;`.
- Trim the text if `trimTextNodes=”true”` and no `xml:space=”preserve”` is in context.

`process_whitespace()`
- Output the whitespace characters.

`process_significant_whitespace()`
- Output the significant whitespace characters.

The methods mentioned above use internal methods to split up the work involved:

`process_namespaces()`
• Invoke `add_namespaces()`.
• Inclusive or exclusive processing of each namespace in context.
Rewrite the prefixes when the parameter `prefixRewrite` is either set to `sequential` or `derived`.
• Sort the namespaces to be output if `sortAttributes="true"`.
• Return a list of namespaces to be output.

`add_namespaces()`
• Add the namespace declarations of the current start element tag to the namespace context.
• Replace undefined characters in the namespace URI with their numeric character references, convert `&`, `<`, `>`, and `"` to `&amp;`, `&lt;`, `&gt;`, and `&quot;`. Quote the result using double quotes.

`remove_namespaces()`
• Remove the deprecated namespaces from the namespace context, i.e. namespaces of elements which occured deeper in the document structure than the current depth.

`process_attributes()`
• Process the attributes of the current start element tag.
• Calculate the current value of `xml:base` if `xmlAncestors="inherit"`.
• Process the simple inerhitable attributes `xml:lang` and `xml:space` if `xmlAncestors="inherit"`.
• Convert `&`, `<`, `>`, and `"` to `&amp;`, `&lt;`, `&gt;`, and `&quot;`. Quote the result using double quotes.
• Replace all occurences of characters `#x9`, `#xA`, and `#xD` with character references `&#x9;`, `&#xA;`, and `&#xD;`.
• Sort the attributes by URI and name if `sortAttributes="true"`.
• Return the list of attributes to be output.

`current_element_visibly_utilizes_prefix?(prefix)`
• Determine if the current element has a qualified name that uses the given prefix.
• Determine if an attribute of the current element has a qualified name that uses the given prefix.

`process_base_uri(attribute_value)`
• Calculate the new `xml:base` URI from the current value of `xml:base` and the given new value of `xml:base`.

2.5. Results

We based our implementation on Ruby and libxml-ruby. libxml-ruby uses libxml2 which provides three different parsers, a DOM parser, a SAX parser, and a StAX parser. Besides operating completely different to each other the parsers provide a different set of functionality to the programmer. The StAX parser (`LibXML::XML::Reader`) provides support for processing DTD information and can add default attributes and expand entities. The SAX parser (`LibXML::XML::SaxParser`) does not provide this functionality. The implementer has to manually process the DTD information.
Furthermore, libxml-ruby does not expose the full functionality of the libxml2 SAX parser. `LibXML::XML::SaxParser` does not deliver the namespace prefixes of an attribute to the class which implements the `LibXML::XML::SaxParser::Callbacks` methods. In order to achieve this, we had to extend the implementation of the class on the C layer. The patch is documented in Appendix A.
Another shortcoming is the absence of options to configure `LibXML::XML::SaxParser`. The class can only parse XML documents which do not contain a structure deeper than 256. When the class parses an XML document with a structure deeper than 256, it cancels the parsing process. This safety belt can be turned off in `LibXML::XML::Reader` with the option `LibXML::XML::Parser:Options::HUGE`.

The Canonical XML Version 2.0 Editor’s Draft focusses on the DOM-based implementation of the *canonicalization method*. Explanations and ideas to the streaming approach with a SAX or StAX parser are scarce. This made it difficult to adapt the hints from the data model, the processing mode, and the pseudocode. The data model of the Editor’s Draft is only suited for the DOM-based approach and not applicable to the streaming approach. The Editor’s Draft published on August 30, 2010 uses the XML Signature Streaming Profile of XPath 1.0 for the data model in a streaming implementation.

3. Performance Evaluation

In this section the performance of the presented implementation of the Canonical XML Version 2.0 Editor's Draft using Ruby 1.9 and libxml-ruby is evaluated. We analyze the impact of each parameter and compare the implementation with the implementation of the former Recommendations provided by libxml2.

This section only discusses the implementation based on `LibXML::XML::Reader` as the class `LibXML::XML::SaxParser` is limited to processing XML documents with a maximum depth of 256 and thus cannot process most of the test files presented below.

3.1. Setup

The performance evaluation uses two criteria: the time period to fully execute the evaluated program and the maximum size used in the main memory (stack and heap). We collect three samples for each testfile and configuration and calculate the average value from that samples. The output is redirected to `/dev/null` so that it does not distort the samples too much. We measure the time and memory usage necessary for processing and not for printing the output.

The UNIX tool `time` can measure the time period needed to execute a given program. Listing 9 shows the output of `time`. The component `real` is the actual time period the process needed to execute. `user` and `sys` refer to the CPU time of that process. The value of `real` is used for the evaluation.

```
$ time -p ./evaluated_program inputfile.xml > /dev/null
real 0.01
user 0.01
sys 0.00
```

Listing 9: `time`

Valgrind is a collection of tools to profile programs. We use the heap profiler *Massif* which has an option to profile the stack usage as well. It takes snapshots of the heap and stack at various points during the execution of the program and writes the output into a file. The file can then be viewed with the tool `ms_print`. It displays a graph showing the amount of memory usage over time. It also shows the maximum value of the memory used by the process.

```
$ valgrind --tool=massif --stacks=yes evaluated_program inputfile.xml > /dev/null

$ ms_print massif.out.7425
--------------------------------------------------------------------------------
Command:            c14n_no_comments flat_simple_100kb.xml
Massif arguments:   (none)
ms_print arguments: massif.out.7425
--------------------------------------------------------------------------------


    MB
1.443^                                                                    #
     |                                                                    #
     |                                                       @:::::@:::::@#
     |                             @::@::@@:::::::::::::::::@:::::@:::::@#
     |                          :::@::@: @ :: :::: : :: ::::@:::::@:::::@#
     |                         ::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:
     |                        :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:
     |                     ::::::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:
     |                   :::: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#::
     |                  :: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#::
     |                 ::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#::
     |               ::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#::
     |             ::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#::
     |           ::::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#::
     |         ::: ::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:::
     |        :: : ::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:::
     |       ::: : ::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:::
     |    :::::: : ::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:::
     |   ::: ::: : ::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:::
     | ::::: ::: : ::::::: :: :::: @::@: @ :: :::: : :: ::::@:::::@:::::@#:::
   0 +----------------------------------------------------------------------->Mi
     0                                                                   17.86
```

Listing 10: Usage of Valgrind

Because the programs evaluated with *Massif* run about twenty times slower than under normal circumstances, the measurement of time period and memory usage are performed in succeeding steps. This allows `time` to capture the time period when the program is executed without performance penalties and *Valgrind* can bear any time-related performance penalties because the only relevant criteria in this evaluation step is the memory being used.

The measured time period relies on the system's current load and can vary depending on how many other processes the system has to execute. To compensate for this the time period is measured three times and the average value is calculated. To complete the measurement in a timely manner, we cancelled a particular test when its execution time exceeded the threshold of 1000 seconds. In this case the memory usage of this particular test was omitted as well.

The memory usage measured with *Valgrind* does not depend on the load of the system. Therefore, we performed the measurement of the memory usage only once.

3.1.1. Testfiles

We used different XML files of various sizes to measure speed and memory footprint of the tested implementations. Every file was created in a version of size 100kB, 1MB, 10MB, 100MB, and 1GB. The files shown below are slightly re-formatted for better readability.

```
<DUMMY><root>
      <element/>
      ...
      <element/>
<root></DUMMY>
```

Listing 11: flat_simple.xml

```
<DUMMY><root>
  <element>
    <element>
      ...
    </element>
  </element>
<root></DUMMY>
```

Listing 12: deep_simple.xml

```
<DUMMY><root xmlns="http://example.com">
  <prefix0:element xmlns:prefix0="http://example0.com"/>
  ...
  <prefixN:element xmlns:prefixN="http://exampleN.com"/>
<root></DUMMY>
```

Listing 13: flat_complex.xml

```
<DUMMY><root xmlns="http://example.com">
  <prefix0:element xmlns:prefix0="http://example0.com">
    <prefix1:element xmlns:prefix1="http://example1.com">
      ...
    </prefix1:element>
  </prefix0:element>
<root></DUMMY>
```

Listing 14: deep_complex.xml

```
<DUMMY><root attrN="N" attrN-1="N-1" ... attr0="0"/></DUMMY>
```

Listing 15: attributes.xml

```
<DUMMY><root xmlns="http://example.com"
             xmlns:ns0="http://example0.com"
             xmlns:ns1="http://example1.com"
             ...
             xmlns:nsN="http://exampleN.com">
  <element ns0:attr0="0"/>
  <element ns1:attr1="1"/>
  ...
  <element nsN:attrN="N"/>
</root></DUMMY>
```

Listing 16: exclusive_namespaces_flat.xml

```
<DUMMY><root xmlns="http://example.com"
             xmlns:ns0="http://example0.com"
             xmlns:ns1="http://example1.com"
             ...
             xmlns:nsN="http://exampleN.com">
  <element ns0:attr0="0">
    <element ns1:attr1="1"/>
      ...
    </element>
  </element>
</root></DUMMY>
```

Listing 17: exclusive_namespaces_deep.xml

```
<DUMMY><root>
  <element>
    This is text content
  </element>
  ...
  <element>
    This is text content
  </element>
<root></DUMMY>
```

Listing 18: text.xml

```
<DUMMY><root>
  <element>
    <!-- This is a comment -->
  </element>
  ...
  <element>
    <!-- This is a comment -->
  </element>
<root></DUMMY>
```

Listing 19: comment.xml

3.1.2. Hardware

We performed the tests on a dedicated machine with Ubuntu Server 10.04.1 LTS installed. The machine contained the following hardware components:

- 16x Intel(R) Xeon(R) CPU E5620 @ 2.40GHz
- 32GB main memory
- Western Digital WDC WD1602ABKS-1 hard disk

3.1.3. Ruby

We chose Ruby as the programming language because it provides a sophisticated class library and data structures to work with and is easy to use. The manipulation of strings is much more comfortable in Ruby than in C.

But Ruby also has its shortcomings. The biggest problem is its performance. A programm written in C (or Java) usually perfroms better than a program written in Ruby.

Ruby 1.8 and earlier used a relatively slow interpreter. Ruby 1.9 uses, YARV, the Yet Another Ruby VM [YARV]. Ruby 1.9 programs are not interpreted, but virtual machine instructions are created from the program and executed in YARV. Ruby 1.9 is considered one of the faster modern scripting languages and faster than Ruby 1.8.

3.2. Impact of Each Parameter

In this section we evaluate the impact of the different parameters on both speed and memory consumption. Every configuration for a specific parameter is tested against the configuration which uses minimal processing (see Listing 20).

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=none --sortAttributes=no --xmlAncestors=none
> /dev/null
```

Listing 20: Invoking Ruby c14n2 with minimal processing

3.2.1. sortAttributes

The impact of the parameter *sortAttributes* was measured with the test file `attributes.xml`.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=none --sortAttributes=yes --xmlAncestors=none
> /dev/null
```

Listing 21: Invoking Ruby c14n2 with attribute sorting

The file `attributes.xml` contains many attributes on a single element in reverse lexicographic order (see Listing 15). The impact of the parameter *sortAttributes* is best shown in such a worst-case szenario. The impact on speed is heavy when using the 100kB file, but less heavy when using the 1MB file. Sorting the attributes takes more time and memory than not sorting at all. The impact on memory usage is less serious, however. The 10MB, 100MB, and 1GB version of attributes.xml could not be processed within the time period of 1000 seconds mentioned earlier.

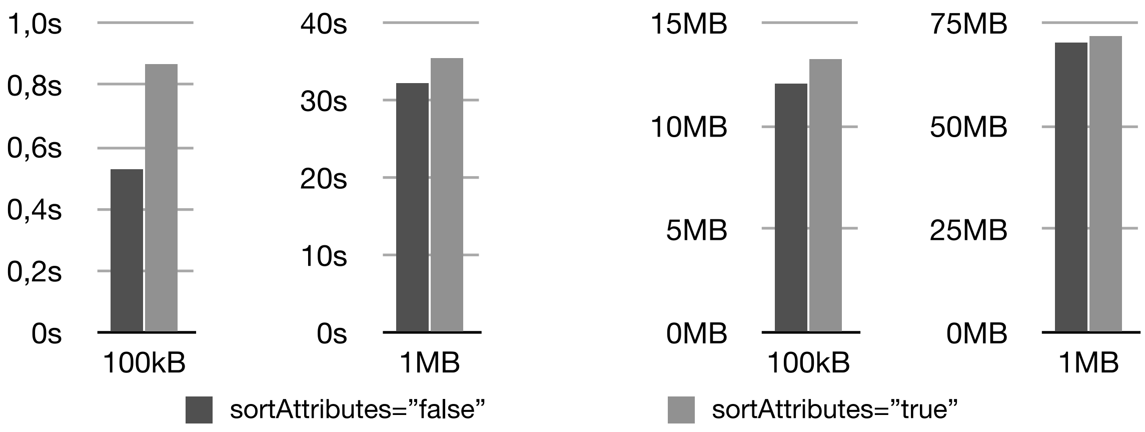


Figure 16: attributes.xml (Time & Memory)

3.2.2. ignoreComments

The impact of the parameter *ignoreComments* was measured with the test file `comments.xml`.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=no --trimTextNodes=no
--prefixRewrite=none --sortAttributes=no --xmlAncestors=none
> /dev/null
```

Listing 22: Invoking Ruby c14n2 with comment processing

The file `comments.xml` contains many elements in a flat hierarchy. Each element contains a comment (see Listing 19). Processing this file is faster with `ignoreComments=”false”`, but the memory usage is constant, regardless of the parameter value. The impact of the parameter increased with the size of the file. The impact on speed with the 100kB file is strange, though. The result indicates that the speed is better with `ignoreComments=“true”`. However, this should not be the case. Not processing the comments should always result in more speed, as the results for the 1MB, 10MB, 100MB, and 1GB version show.

`comments.xml` contains a flat structure and the streaming-based implementation does not need to store context information such as namespaces or XML attributes when processing such a structure. Therefore, the memory usage is constant among the different file sizes.

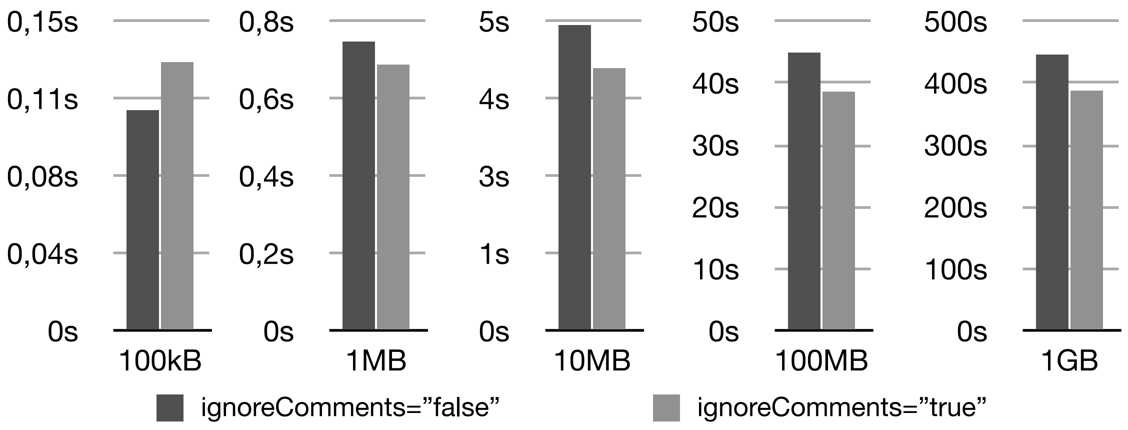


Figure 17: comments.xml (Time)

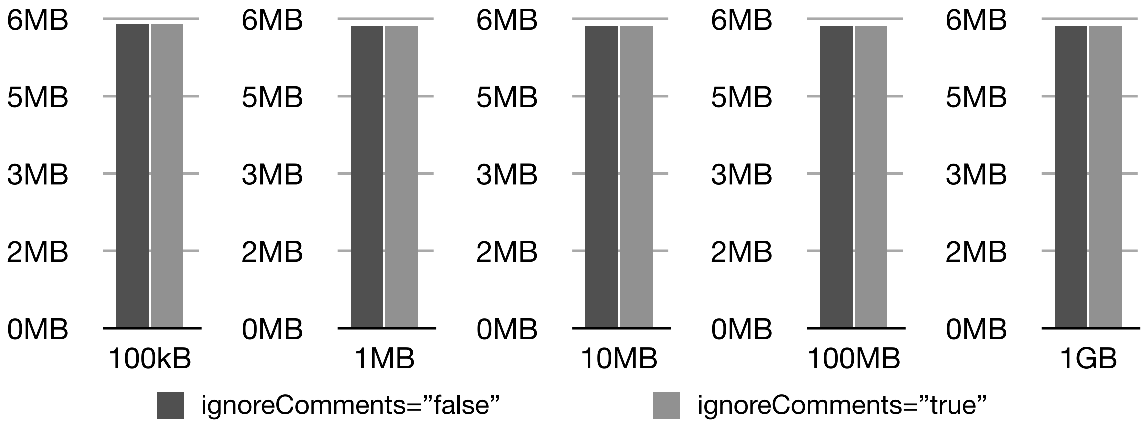


Figure 18: comments.xml (Memory)

3.2.3. exclusiveMode

The impact of the parameter *exclusiveMode* was measured with the test files `exclusive_namespaces_flat.xml` and `exclusive_namespaces_deep.xml`.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=yes --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=none --sortAttributes=no --xmlAncestors=none
> /dev/null
```

Listing 23: Invoking Ruby c14n2 with exclusive mode

The file `exclusive_namespaces_flat.xml` contains elements in a flat hierarchy. Each element visibly utilizes a distinct namespace defined in the enclosing element (see Listing 16). The impact on both speed and memory usage is serious. Processing a file with many namespace declarations in exclusive mode is more expensive than processing the file in inclusive mode, because the *canonicalization method* has to check every namespace declaration in the context if it is visibly utilized by the current element. Processing the file in exclusive mode requires more memory. Processing the 10MB, 100MB, and 1GB version of `exclusive_namespaces_flat.xml` lasted more than 1000 seconds and was cancelled.

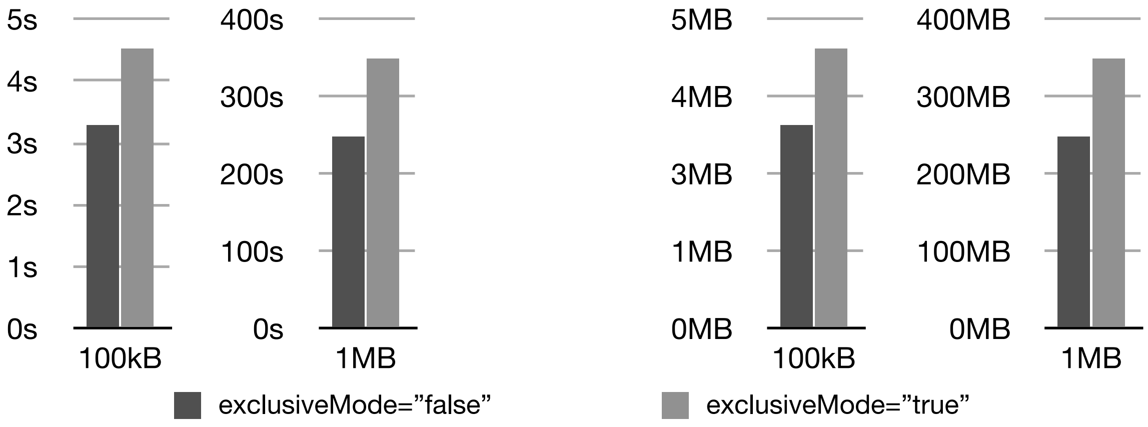


Figure 19: exclusive_namespaces.flat.xml (Time & Memory)

The file `exclusive_namespaces_deep.xml` contains elements in a deeply nested hierarchy. Each element visibly utilizes a distinct namespace defined in the enclosing element (see Listing 17). Similar to the test with the file `exclusive_namespaces_flat.xml`, the impact of the parameter on speed is also significant when using a file with a deeply nested structure. Our implementation needs a little more memory in exclusive mode when processing the 1MB and 100MB file, but uses a little less memory when processing the 100kB and 10MB version of the file. The 1GB version of `exclusive_namespaces_deep.xml` was omitted because the test did not complete within 1000 seconds.

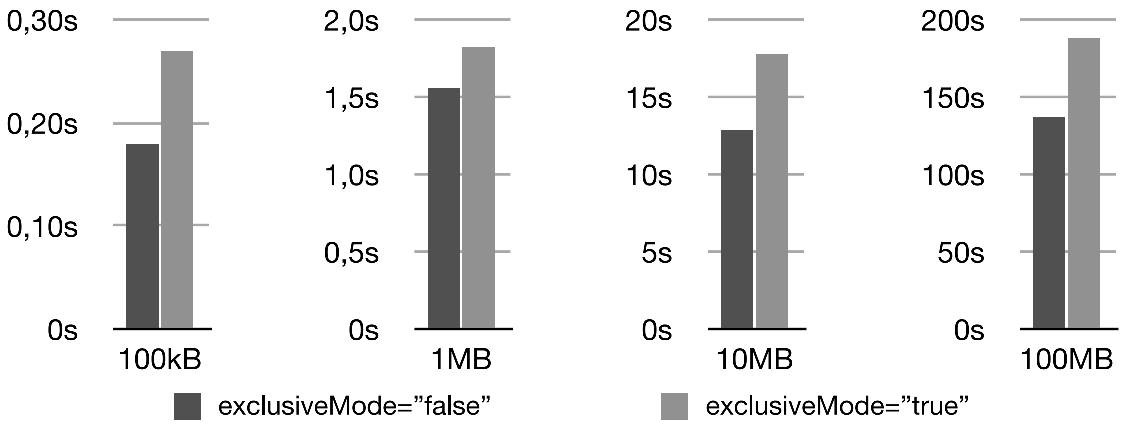


Figure 20: exclusive_namespaces_deep.xml (Time)

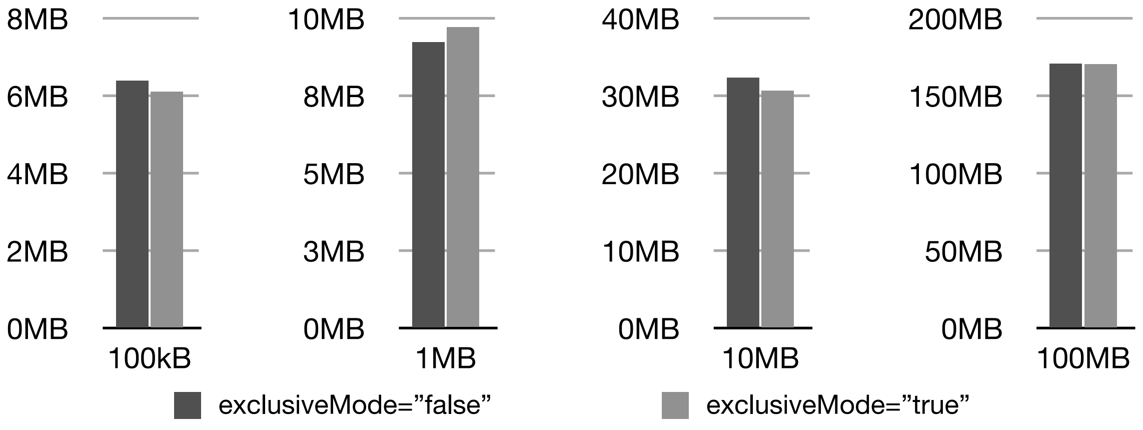


Figure 21: exclusive_namespaces_deep.xml (Memory)

3.2.4. trimTextNodes

The impact of the parameter *trimTextNodes* was measured with the test file `text.xml`.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=yes --trimTextNodes=yes
--prefixRewrite=none --sortAttributes=no --xmlAncestors=none
> /dev/null
```

Listing 24: Invoking Ruby c14n2 with trimming of text nodes

`text.xml` contains many elements in a flat hierarchy. Each element contains text (see Listing 18). Trimming the text has a slight impact when processing the larger files. The impact is permissible when processing the smaller files. Due to the flat structure of `text.xml` the memory usage is constant among the processing of different file sizes. The canonicalization method does not have to store context information such as namespaces or xml attributes in these cases.

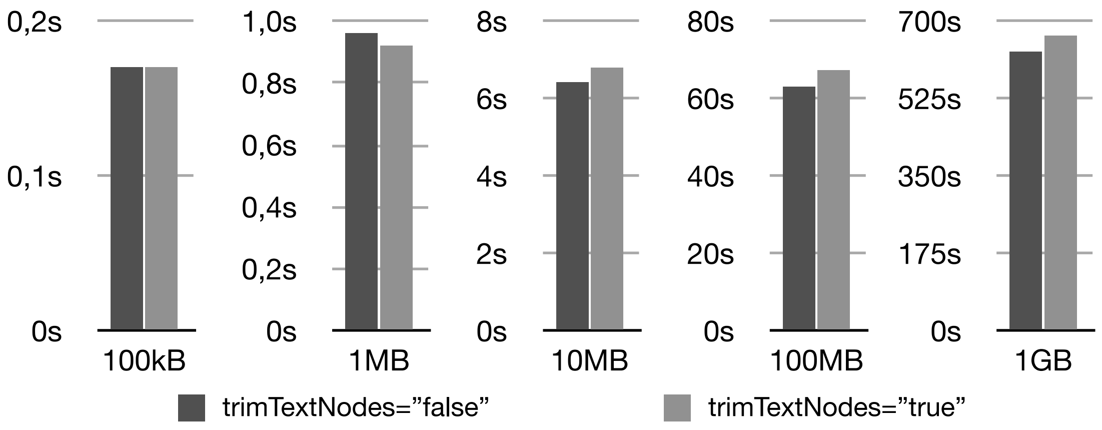


Figure 22: text.xml (Time)

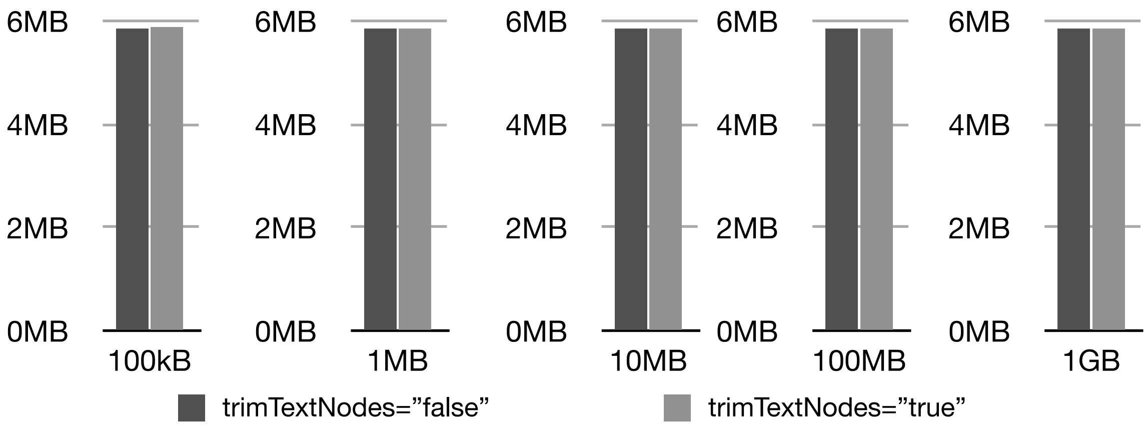


Figure 23: text.xml (Memory)

3.2.5. xmlAncestors

The impact of the parameter *xmlAncestors* was measured with the test file `attributes.xml`.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=none --sortAttributes=no --xmlAncestors=inherit
> /dev/null
```

Listing 25: Invoking Ruby c14n2 with inheriting xml ancestors

The file does not contain attributes in the XML namespace, so the results with other files which contain those attributes may vary. However, when the parameter *xmlAncestors* is set to “inherit”, the canonicalization method has to check every attribute if its qualified name is `xml:base`, `xml:space`, or `xml:lang` instead of just processing the attribute. The memory usage is slightly higher with `xmlAncestors="inherit"`, because there are more String comparisons involved than with `xmlAncestors="none"`. The tests with the 10MB, 100MB, and 1GB version of `attributes.xml` did not complete within 1000 seconds and were cancelled.

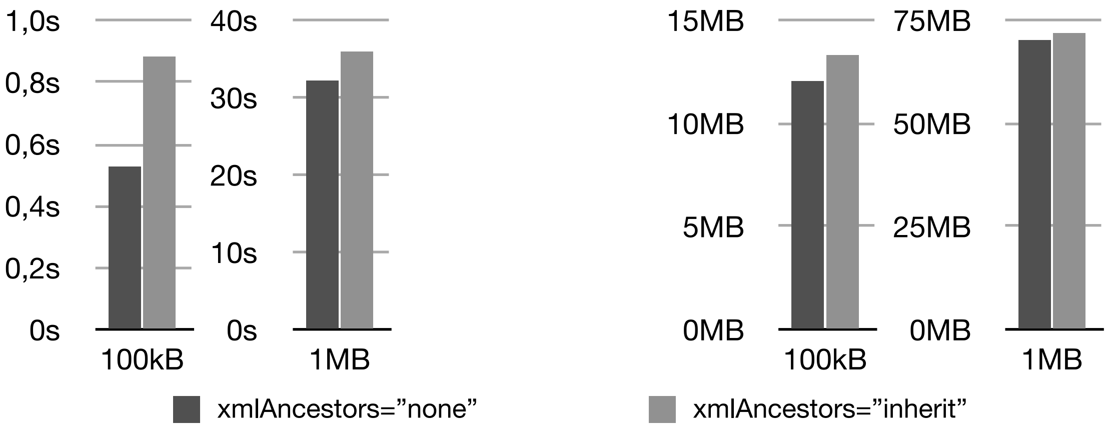


Figure 24: attributes.xml (Time & Memory)

3.2.6. prefixRewrite

The impact of the parameter *prefixRewrite* was measured with the test files `flat_complex.xml` and `deep_complex.xml`.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=sequential --sortAttributes=no --xmlAncestors=none
> /dev/null
```

Listing 26: Invoking Ruby c14n2 with sequential prefix rewriting

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=derived --sortAttributes=no --xmlAncestors=none
> /dev/null
```

Listing 27: Invoking Ruby c14n2 with derived prefix rewriting

The file `flat_complex.xml` contains a flat structure with many elements. Each element contains a distinct namespace declaration and uses the declared namespace in its qualified name (see Listing 13). The parameter value directly influences the speed of the *canonicalization method*, "none" being the fastest and "derived" being the slowest option. The memory footprint is approximately the same among the different values. Sequential prefix rewriting increments a single counter value based on the occuring namespace declarations. Derived prefix rewriting computes a SHA-1 hash value from a namespace URI. Both parameters do not store additional values during the processing when compared to the no prefix rewriting option. The tests with the 1GB version of `flat_complex.xml` did not complete within 1000 seconds and were cancelled.

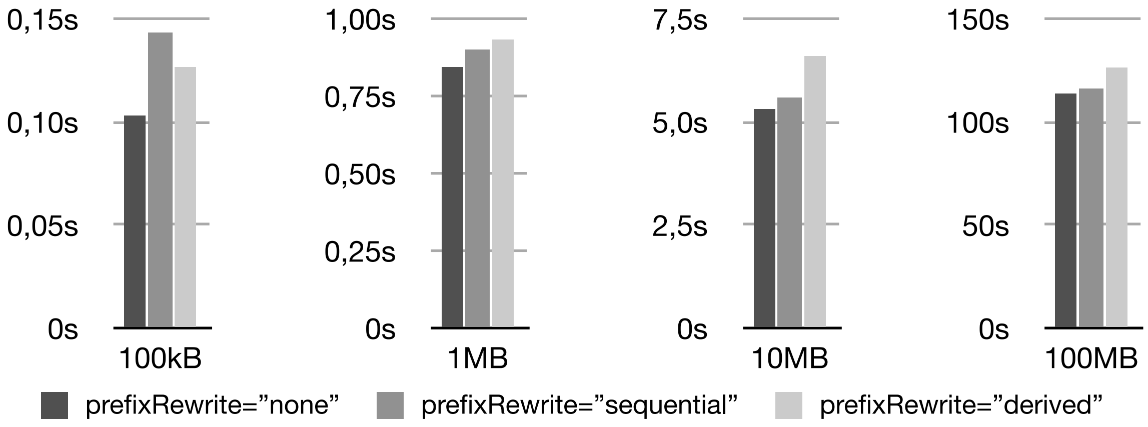


Figure 25: flat_complex.xml (Time)

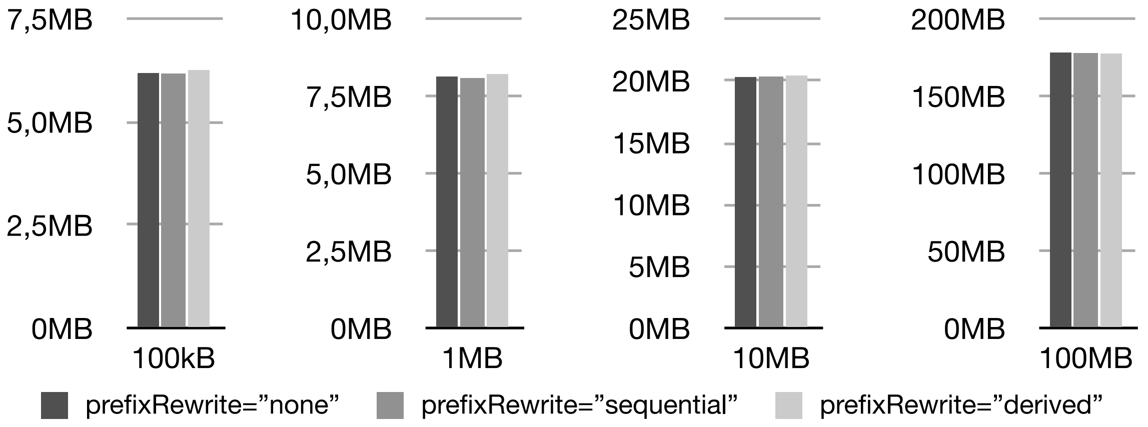


Figure 26: flat_complex.xml (Memory)

The file `deep_complex.xml` contains a deeply nested structure with many elements. Each element contains a distinct namespace declaration and uses the declared namespace in its qualified name (see Listing 14). Processing the 1MB version of the file was faster with derived prefix rewriting than with sequential prefix rewriting and with no prefix rewriting. In normal situations, `prefixRewrite=”none”` should perform faster than `prefixRewrite=”sequential”`. `prefixRewrite=”derived”` should perform slower than the other two options because computing a SHA-1 usually takes longer than not doing any processing at all or just incerementing a counter value. The memory usage is approximately the same, but increases with the size of the file. The 1GB version of the file could not be processed with the option `prefixRewrite=”sequential”` and `prefixRewrite=”derived”` within 1000 seconds, so these tests were cancelled.

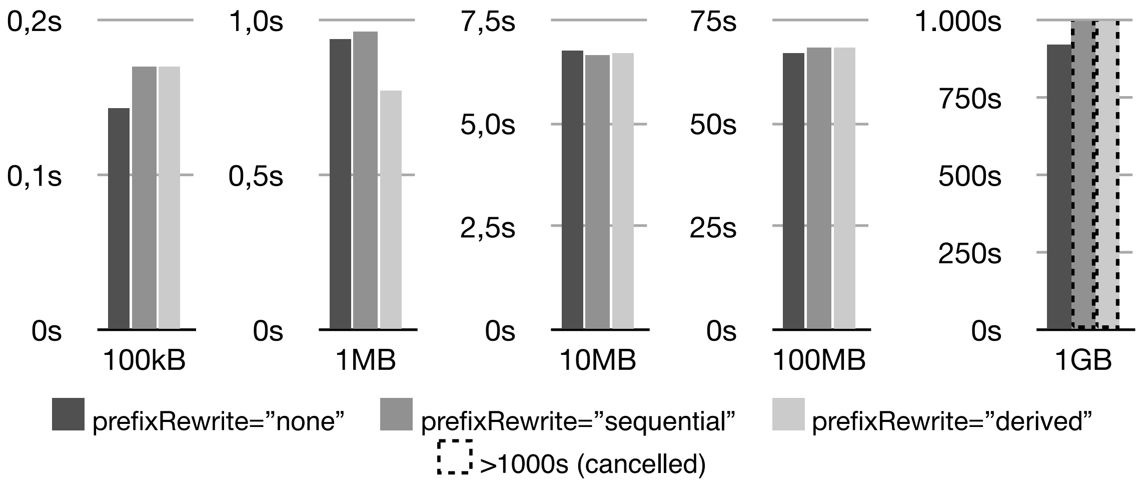


Figure 27: deep_complex.xml (Time)

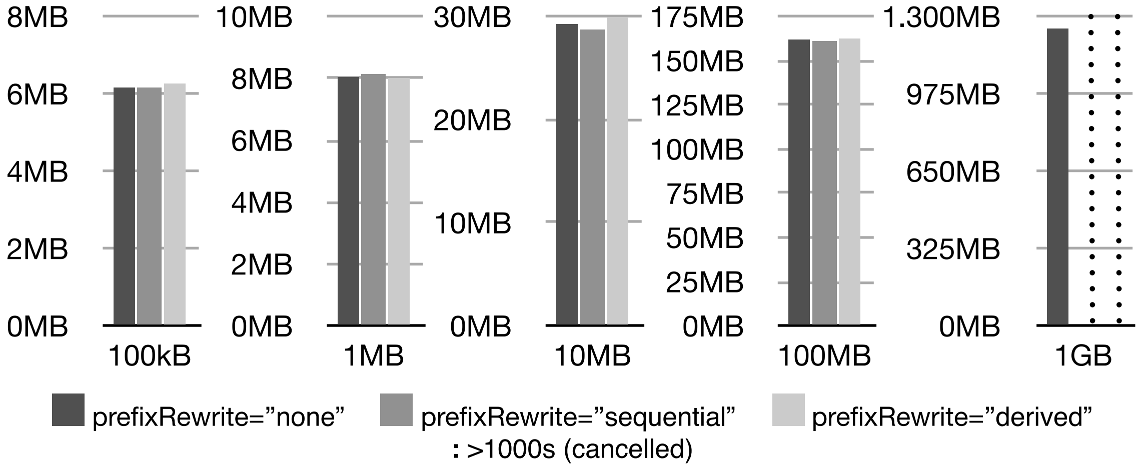


Figure 28: deep_complex.xml (Memory)

3.3. Comparison with libxml2 c14n Implementations

This chapter compares our c14n2 implementation with libxml2's c14n implementations. Processing the comments were disabled for this comparison.

3.3.1. Ruby c14n2 vs. libxml2 c14n

The comparison of our implementation with the c14n version of libxml2 was measured with the test files `flat_complex.xml`, `deep_complex.xml`, `exclusive_namespaces_deep.xml`, and `text.xml`.

Our implementation was invoked in its Canonical XML Version 1.0 mode without processing comments using the command in Listing 20. The output was piped to /dev/null to omit the overhead of printing the output in the terminal.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--ignoreComments=yes --trimTextNodes=no --prefixRewrite=none
--sortAttributes=yes --xmlAncestors=none > /dev/null
```

Listing 28: Invoking Ruby c14n2 in c14n mode

The libxml2 implementation also uses the Canonical XML Version 1.0 Recommendation without processing comments.

```
$ ./c14n_no_comments input_file.xml > /dev/null
```

Listing 29: Invoking libxml2 c14n

libxml2 c14n uses a DOM to canonicalize the input document (see Appendix B for more information on how libxml2 c14n is used).

The overall speed of libxml2 c14n is much better than our Ruby-based implementation, given that the structure of the input file is simple (e.g. `flat_complex.xml` or `text.xml`). In these situations it performs usually much better than our Ruby-based implementation. When the input document contains a deeply nested structure (e.g. deep_complex or exclusive_namespaces_deep.xml), libxml2 performs worse than our implementation - both in speed and memory footprint. libxml2 c14n could not even canonicalize the 100MB and 1GB versions of the file `deep_complex.xml` within 1000 seconds and was cancelled.

If the input document is very large, libxml2 c14n uses much more memory than our implementation. This is the result of the fundamentally different approach:

libxml2 needs a DOM to be created from the input document before being able to canonicalize it. If the input file's size is very large, libxml2 c14n needs a lot of memory (canonicalizing the 1GB version of `text.xml` uses at most approximately 10,6GB).

The 1GB versions of flat_complex.xml and exclusive_namespaces_deep.xml could not be processed by both implementations within 1000 seconds and the tests with these files were cancelled.

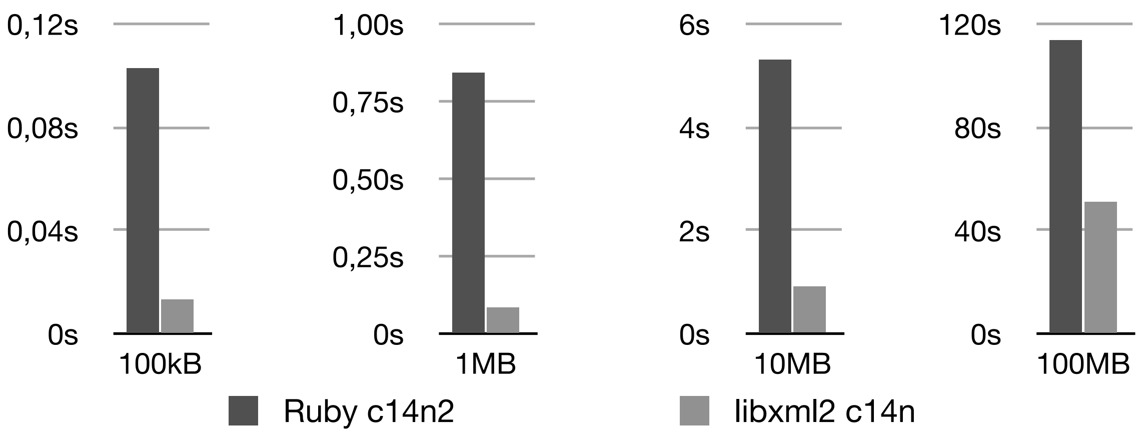


Figure 29: flat_complex.xml (Time)

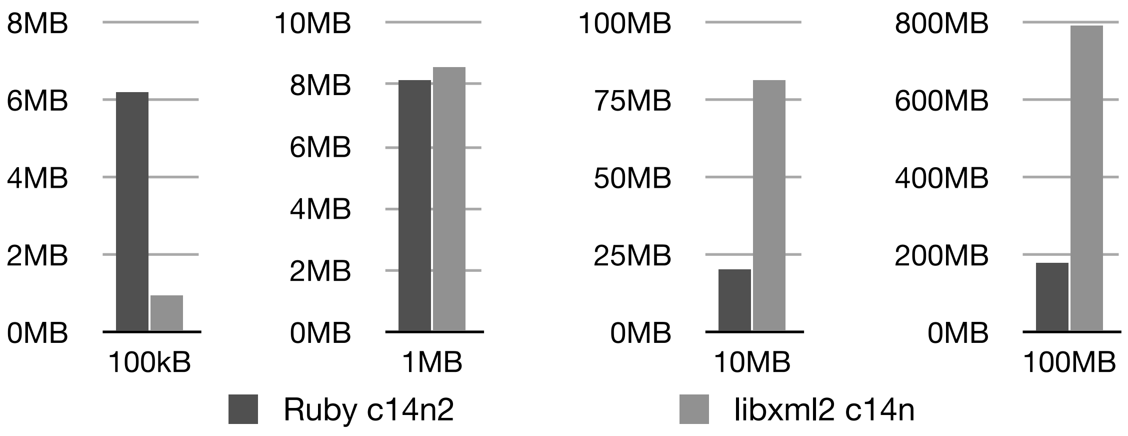


Figure 30: flat_complex.xml (Memory)

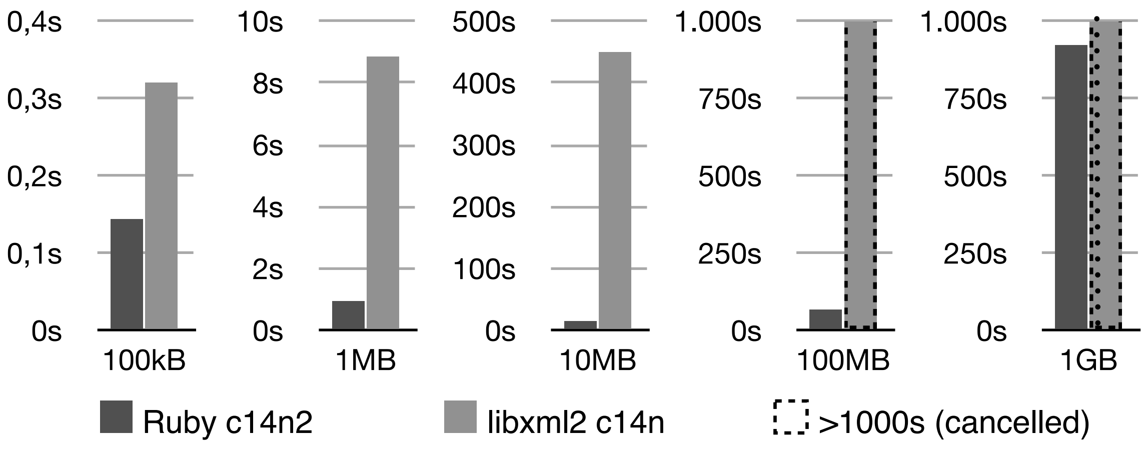


Figure 31: deep_complex.xml (Time)

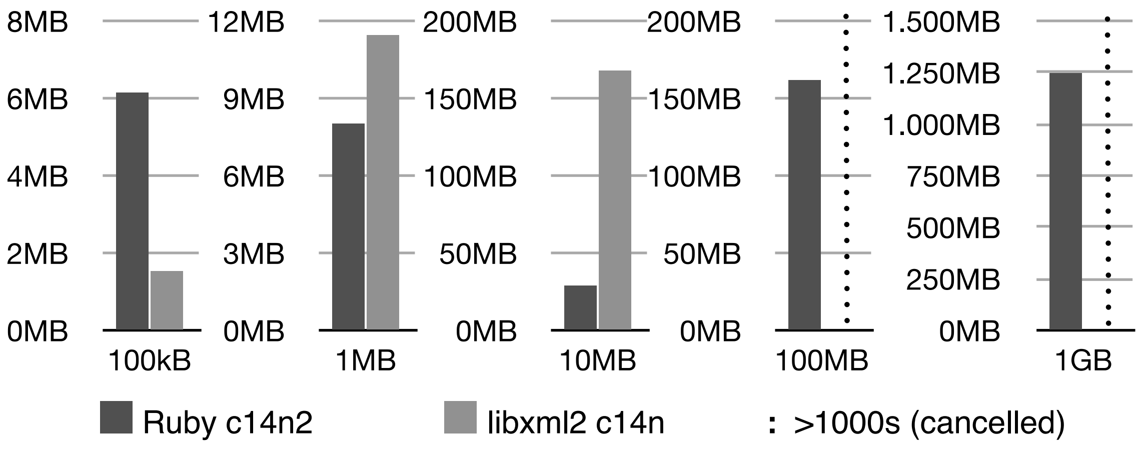


Figure 32: deep_complex.xml (Memory)

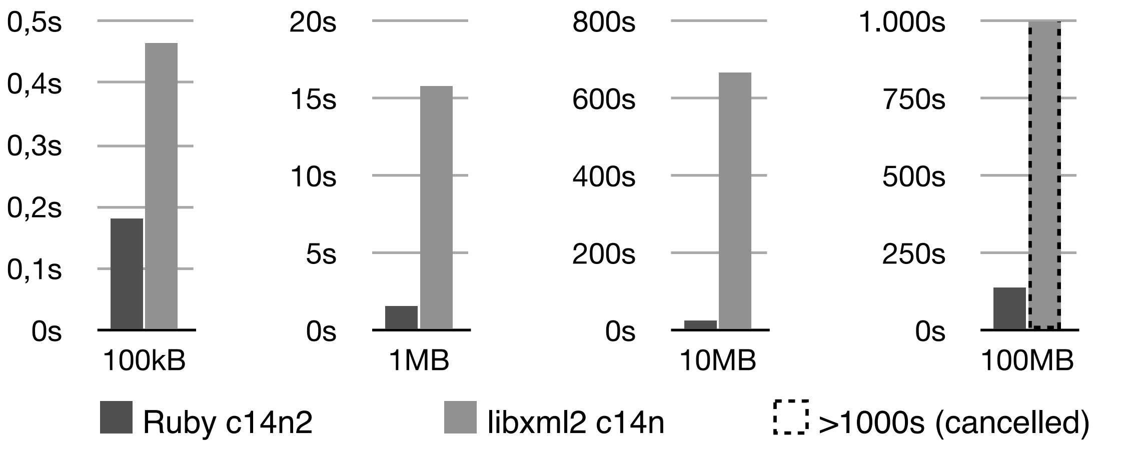


Figure 33: exclusive_namespaces_deep.xml (Time)

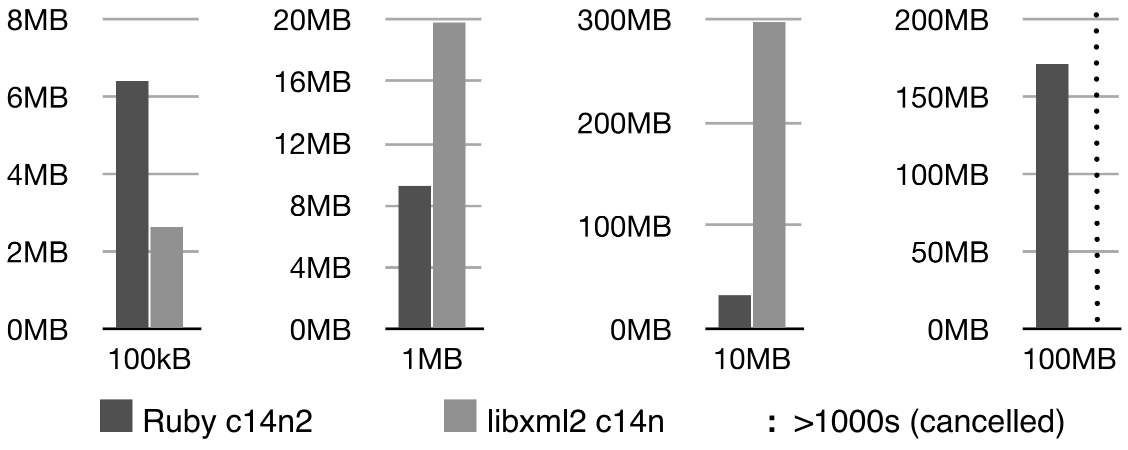


Figure 34: exclusive_namespaces_deep.xml (Memory)

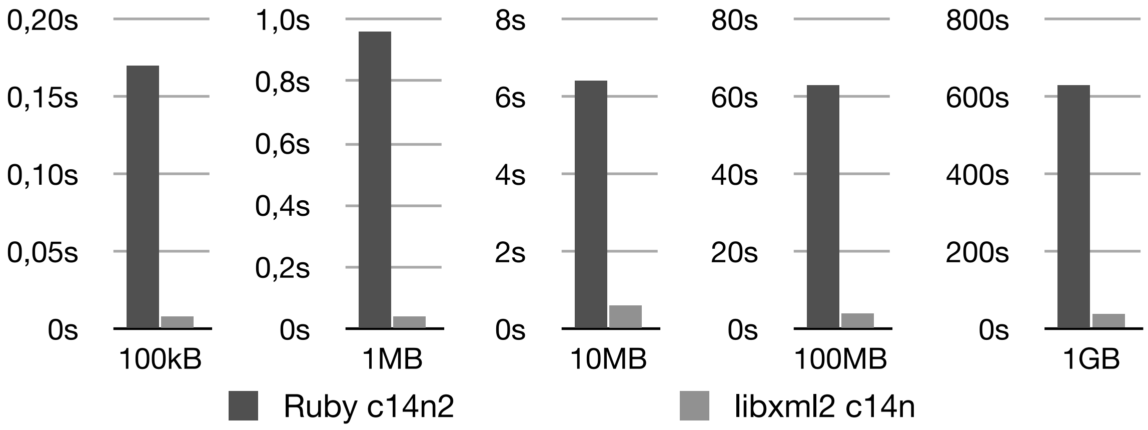


Figure 35: text.xml (Time)

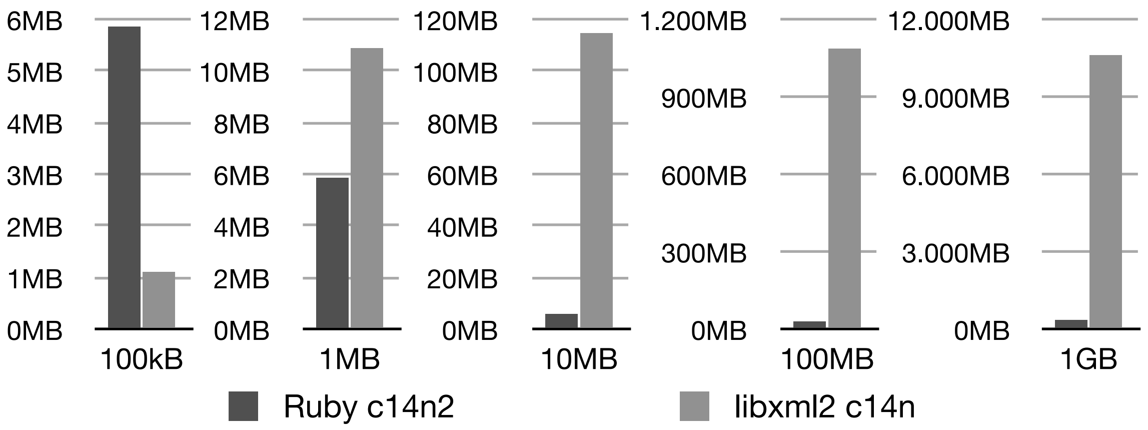


Figure 36: text.xml (Memory)

3.3.2. Ruby c14n2 vs. libxml2 exc-c14n

The comparison of our implementation with the exc-c14n version of libxml2 was measured with the test files `exclusive_namespaces_flat.xml` and `exclusive_namespaces_deep.xml`.

Our implementation was invoked in its Exclusive XML Canonicalization Version 1.0 mode without processing comments using the command in Listing 30.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=yes --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=none --sortAttributes=yes --xmlAncestors=none
> /dev/null
```

Listing 30: Invoking Ruby c14n2 in exc-c14n mode

The libxml2 implementation also uses the Exclusive XML Canonicalization Version 1.0 Recommendation without processing comments.

```
$ ./exc_c14n_no_comments input_file.xml > /dev/null
```

Listing 31: Invoking libxml2 exc-c14n

The libxml2 implementation is considerably faster than our implementation in every test with the different version of `exclusive_namespaces_flat.xml` and `exclusive_namespaces_deep.xml`. It even uses less memory when processing the 100kB and 1MB versions of the two files.

The 10MB, 100MB, and 1GB version of `exclusive_namespaces_flat.xml` could not be processed within the time period of 1000 seconds by both implementations. Our implementation could not process the 1GB version of `exclusive_namespaces_deep.xml` within this time period.

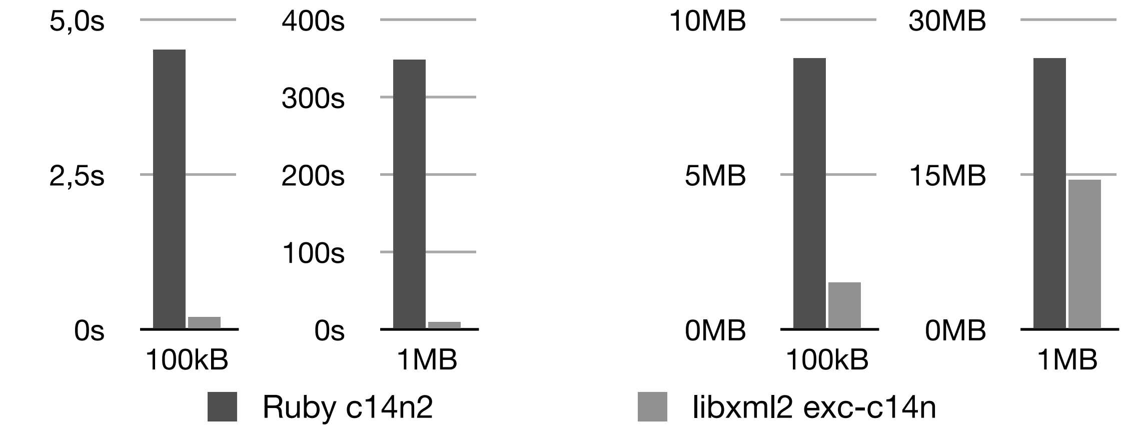


Figure 37: exclusive_namespaces_flat.xml (Time & Memory)

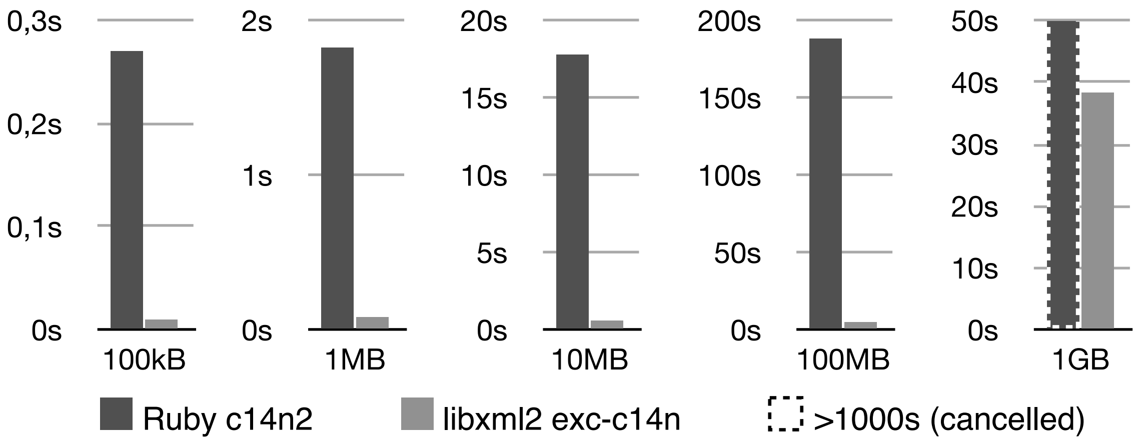


Figure 38: exclusive_namespaces_flat.xml (Time)

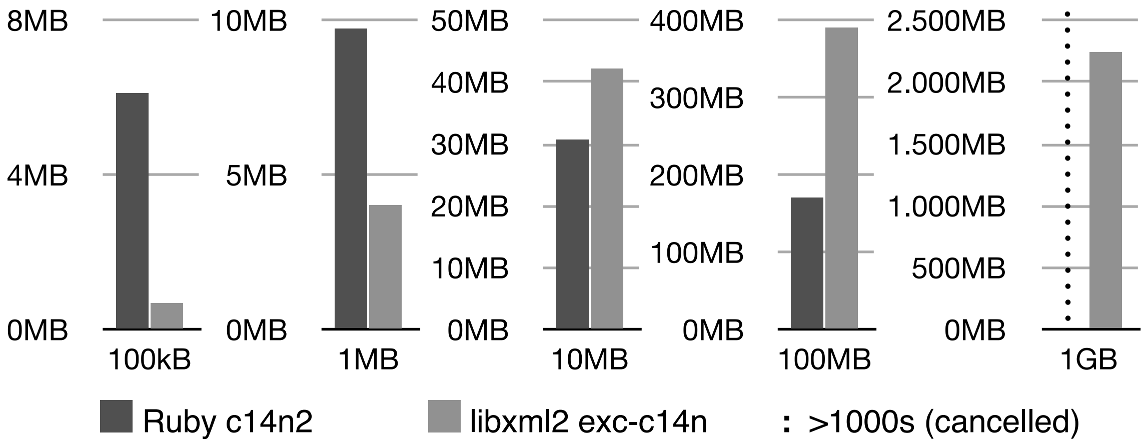


Figure 39: exclusive_namespaces_deep.xml (Memory)

3.3.3. Ruby c14n2 vs. libxml2 c14n11

The comparison of our implementation with the c14n11 version of libxml2 was measured with the file `attributes.xml`. Our implementation was invoked in its Canonical XML Version 1.1 mode without processing comments using the command in Listing 32.

```
$ ruby -w -I lib/ bin/c14n2 --parser=stax --file=input_file.xml
--exclusiveMode=no --ignoreComments=yes --trimTextNodes=no
--prefixRewrite=none --sortAttributes=yes --xmlAncestors=inherit
> /dev/null
```

Listing 32: Invoking Ruby c14n2 in c14n11 mode

The libxml2 implementation also uses the Canonical XML Version 1.1 Recommendation without processing comments.

```
$ ./c14n11_no_comments input_file.xml > /dev/null
```

Listing 33: Invoking libxml2 c14n11

libxml2 canonicalizes the 100kB version of `attributes.xml` in half the time of our implementation, but is just slightly slower when processing the 1MB version of the file. Creating the DOM of a 100kB file seems to be much more efficient than creating the DOM of a 1MB file.
Our implementation uses much more memory than libxml2 c14n11 to canonicalize the 100kB and 1MB version of `attributes.xml`. This is possibly the overhead of the Ruby programming language compared to the usage of the C programming language.
The 10MB, 100MB, and 1GB version of `attributes.xml` could not be processed within 1000 seconds and the tests were cancelled.

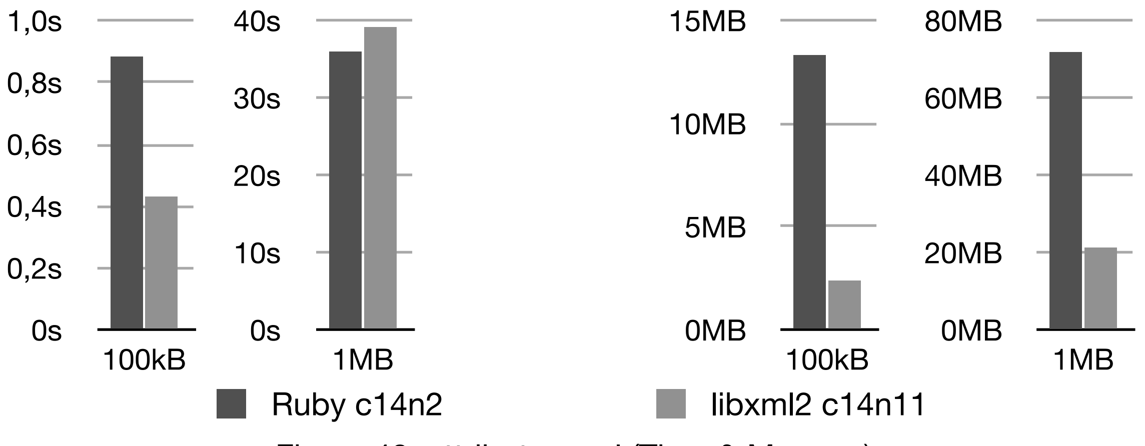


Figure 40: attributes.xml (Time & Memory)

4. Summary

In this thesis, we evaluated the feasibility and performance of the upcoming new major version of the Canonical XML Recommendation, Canonical XML Version 2.0. We implemented the Editor's Draft (22 June 2010) in the Ruby programming language with libxml-ruby, the Ruby language bindings of the popular C library libxml2. libxml-ruby provides three parsers: a tree parser (DOM), a push parser (SAX), and a pull parser (StAX). Two approaches were implemented: one uses the SAX parser, the other one uses the StAX parser. Both approaches are based on streaming and do not require the whole document to be parsed and held in memory before being able to canonicalize it.

Our two approaches differ slightly from each other. The StAX-based approach uses the binding for the pull parser of libxml2, `LibXML::XML::Reader`. It provides access to all information of the input document and because it is a validating parser, it handles DTD information such as default attributes and entity references by itself. The SAX-based approach uses the binding for the non-validating push parser of libxml2, `LibXML::XML::SaxParser`. This parser has some limitations. The parser does not process DTD information automatically but only pushes the related events to its delegate. The implementer has a lot of additional work to do with implementing the functionality related to DTD information when using this parser. Another shortcoming is the omission of namespace prefixes of attributes. The libxml2 SAX2 parser delivers the namespace prefixes to the registered callback function but the Ruby binding of this parser does not deliver the namespace prefixes to the Ruby layer. The source code of libxml-ruby reveals that the developers of libxml-ruby simply did not implement the passing of namespace prefixes from the C layer to the Ruby layer. The single most serious shortcoming of this parser is its inability to process XML documents which contain a hierarchy deeper than 256. The pull parser supports an option which allows the processing of hierarchies deeper than 256. The push parser lacks in this option.

The Editor's Draft of June 22, 2010 does not contain a description of the input model for a streaming implementation of the canonicalization method. As a result, the implementation of the selection mechanism is not part of this thesis. During the work on this thesis a new Editor's Draft and a new Working Draft were released. They use the XML Signature Streaming Profile of XPath 1.0 to describe the input model. Adding the selection mechanism to our implementation should be possible with few modifications to the existing architecture and codebase.

The performance evaluation of the different parameters with the StAX-based implementation exposed a negative performance impact with `exclusiveMode=”true”` and `sortAttributes=”yes”`. The impact of the parameters `prefixRewrite` and `trimTextNodes` in our tests was lower than expected.

After analyzing the impact of the parameters we compared the performance of our StAX-based implementation to the implementation of libxml2 which supports Canonical XML Version 1.0, Exclusive XML Canonicalization Version 1.0, and Canonical XML Version 1.1. The libxml2 versions process small XML documents considerably faster than our implementation. The reason is the difference in execution speed between an extremely fast and mature C implementation and the relatively slow and not optimized implementation in Ruby.

Our implementation often performs better than the libxml2 implementation when processing large documents. The libxml2 version uses the DOM-based approach. Creating the DOM of a very large input document needs time and memory. Some documents cannot be canonicalized because the machine's amount of main memory (and swap space) limits the size available to create the DOM. The machine in this work has 32GB of RAM but not every machine which performs XML canonicalization has that much memory. Our implementation uses the streaming-based approach which in most cases uses less memory when processing large files than an implementation which uses the DOM-based approach.

To leverage the best of both worlds, we suggest to implement a production quality version of the Canonical XML Version 2.0 *canonicalization method* with a programming language faster than Ruby. A very fast programming language like C combined with a mature XML library like libxml2 and the advantages of a streaming XML parser would be a good combination. The implementation could then provide bindings to scripting languages like Ruby with the help of the *Ruby C Language API.* This would allow for a much faster execution speed than our implementation but retain the comfort of using a very high-level programming language like Ruby.

A. libxml-ruby Patch

The patch for libxml-ruby changes the way attributes and namespaces are delivered to the class which includes `LibXML::XML::SaxParser::Callbacks`. Originally, attributes are provided as an instance of the class Hash with the local name as the key and the attribute's value as value. Namespaces are provided as a Hash with the prefix as key and the namespace's URI as value.

After applying the patch, attributes and namespaces are provided as Arrays which contain a Hash for every attribute or namespace. A Hash stores the values of an attribute for the keys `localname`, `prefix`, `value`, `uri`. A Hash which stores a namespace stores the values for the keys `prefix` and `uri`. With this solution the full information about a start-element is reported in the invoked callback.

The patch is generated using the command line utiliy `diff`:

```
$ diff ruby_xml_sax2_handler_orig.c ruby_xml_sax2_handler_patched.c
```

```
197,198c197,204
<   VALUE attributes = rb_hash_new();
<   VALUE namespaces = rb_hash_new();
---
>   VALUE attributes = rb_ary_new2((long)nb_attributes);
>   VALUE namespaces = rb_ary_new2((long)nb_namespaces);
>
>   /* Keys for the values in namespace and attribute Hash */
>   VALUE localname_key = ID2SYM(rb_intern("localname"));
>   VALUE prefix_key = ID2SYM(rb_intern("prefix"));
>   VALUE value_key = ID2SYM(rb_intern("value"));
>   VALUE uri_key = ID2SYM(rb_intern("uri"));
209c215
<       VALUE attrName = rb_str_new2(xattributes[i+0]);
---
>       VALUE attrLocalname = rb_str_new2(xattributes[i+0]);
211,212c217,218
<       /* VALUE attrPrefix = xattributes[i+1] ? rb_str_new2
(xattributes[i+1]) : Qnil;
<          VALUE attrURI = xattributes[i+2] ? rb_str_new2(xattributes
[i+2]) : Qnil; */
---
>       VALUE attrPrefix = xattributes[i+1] ? rb_str_new2(xattributes
[i+1]) : Qnil;
>       VALUE attrURI = xattributes[i+2] ? rb_str_new2(xattributes[i
+2]) : Qnil;
214c220,228
<       rb_hash_aset(attributes, attrName, attrValue);
---
>       /* Store attribute information in a Hash */
>       VALUE attribute = rb_hash_new();
>       rb_hash_aset(attribute, localname_key, attrLocalname);
>       rb_hash_aset(attribute, prefix_key, attrPrefix);
>       rb_hash_aset(attribute, value_key, attrValue);
>       rb_hash_aset(attribute, uri_key, attrURI);
>
>       /* Add attribute to Array of attributes */
>       rb_ary_push(attributes, attribute);
225c239,246
<       rb_hash_aset(namespaces, nsPrefix, nsURI);
---
>
>       /* Store namespace information in a Hash */
>       VALUE namespace = rb_hash_new();
>       rb_hash_aset(namespace, prefix_key, nsPrefix);
>       rb_hash_aset(namespace, uri_key, nsURI);
>
>       /* Add namespace to Array of namespaces */
>       rb_ary_push(namespaces, namespace);
```

Listing 34: Patch for libxml-ruby

B. libxml2's Built-in c14n API

libxml2 provides an API to canonicalize an XML document according to the Recommendations Canonical XML Version 1.0, the Exclusive XML Canonicalization Version 1.0, and the Canonical XML Version 1.1. The API expects a reference to a DOM. The function `xmlC14NDocDumpMemory()` processes the DOM and returns the canonicalized output indirectly. The program then prints the canonicalized document to `stdout`. The CD-ROM contains a projct with the following source file together with a Makefile to produce binaries for the different versions of Canonical XML with and without comments:

```
int main (int argc, const char *argv[]) {
  if (argc < 2) {
    printf("Usage: %s <input file>", argv[0]);
    return 1;
  }

  int parseOptions =
    XML_PARSE_NOENT | XML_PARSE_DTDATTR | XML_PARSE_HUGE;

  xmlDocPtr doc = xmlReadFile(argv[1], NULL, parseOptions);
  if (doc == NULL) {
    printf("Error: could not parse file %s\n", argv[1]);
    return 1;
  }

  xmlChar *outputString = NULL;
  int sucsessful = xmlC14NDocDumpMemory(
                     doc,
                     NULL,
                     MODE,
                     NULL,
                     COMMENTS,
                     &outputString
                   );
  if (!successful) {
    printf("Error: could canonize file %s\n", argv[1]);
    return 1;
  }

  // Output to stdout
  fputs((char *)outputString, stdout);

  xmlFreeDoc(doc);

  // Free the global variables that may have been allocated by the
  //parser.
  xmlCleanupParser();

  return 0;
}
```

Listing 35: Using libxml2's c14n API

C. Content of CD-ROM

- **Paper** - contains files used for this thesis, e.g. diagrams, graphics, etc. The file Studienarbeit.{pages, pdf} is this thesis. The file Performance_Values.{numbers, pdf} contains the raw values gathered in the performance evaluation of this thesis.

- **Performance** - contains the raw files that Valgrind created as the result of its memory usage analysis.

- **Seminar** - contains the presentation slides created for the seminar talks.

- **Source** - contains the source code of different projects:

    - **c14n2** - Implementation of the Canonical XML Version 2.0 Editor's Draft of 22 June, 2010. Copyright Manuel Binna.

    - **LibXML_C14N** - Implementation of a small project using libxml2 canonicalization for the performance evaluation. Copyright Manuel Binna.

    - **libxml-ruby-mbinna-1.1.3** - Patched libxml-ruby library in which the SAX parser delivers the namespace prefixes of attributes from the C layer to the Ruby layer.

    - **Testfiles** - Files used during the development of the C14N2 implementation.

D. References

[C14N]	Canonical XML Version 1.0 W3C Recommendation 15 March 2001 http://www.w3.org/TR/xml-c14n (visited October 11, 2010, at 7:14pm)
[EXC-C14N]	Exclusive XML Canonicalization Version 1.0 W3C Recommendation 18 July 2002 http://www.w3.org/TR/xml-exc-c14n (visited October 11, 2010, at 7:16pm)
[C14N11]	Canonical XML Version 1.1 W3C Recommendation 2 May 2008 http://www.w3.org/TR/xml-c14n11 (visited October 11, 2010, at 7:15pm)
[C14N2]	Canonical XML Version 2.0 W3C Editor’s Draft 22 June 2010
[XMLSIG]	XML Signature Syntax and Processing (Second Edition) W3C Recommendation 10 June 2008 http://www.w3.org/TR/xmldsig-core (visited October 11, 2010, at 7:21pm)
[XMLBASE]	XML Base (Second Edition) W3C Recommendation 28 January 2009 http://www.w3.org/TR/xmlbase (visited October 11, 2010, at 7:22pm)
[XML]	Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C Recommendation 26 November 2008 http://www.w3.org/TR/REC-xml (visited October 11, 2010, at 7:22pm)
[EXI]	Efficient XML Interchange Working Group. Public Page World Wide Web Consortium (W3C) http://www.w3.org/XML/EXI/ (visited July 14, 2010, at 11:38am)

[PICKAXE]	Programming Ruby 1.9 - The Pragmatic Programmer's Guide. Dave Thomas with Chad Fowler and Andy Hunt. The Pragmatic Programmers. 2009.
[LIBXML-RUBY-PR]	LibXml Ruby Project http://libxml.rubyforge.org (visited July 14, 2010, at 0:43 pm)
[LIBXML-RUBY-API]	libxml-ruby API Documentation http://libxml.rubyforge.org/rdoc/index.html (visited July 14, 2010, at 0:30pm)
[TEXTREADER]	Libxml2 XmlTextReader Interface tutorial http://www.xmlsoft.org/xmlreader.html (visited July 14, 2010, at 0:49pm)
[LIBXML2]	libxml - The XML C parser and toolkit of Gnome http://www.xmlsoft.org (visited August 21, 2010, at 7:56pm)
[STREAM_XPATH]	XML Signature Streaming Profile of XPath 1.0 W3C Editor's Draft 22 July 2010 http://www.w3.org/2008/xmlsec/Drafts/xmldsig-xpath (visited August 22, 2010, at 0:51pm)
[RUBY-RDOC]	Ruby RDoc Documentation http://ruby-doc.org/ruby-1.9/index.html (visited September 15, 2010, at 10:27pm)
[YARV]	YARV - Yet Another Ruby VM http://ruby.about.com/od/newinruby191/a/YARV.htm (visited September 27, 2010, at 9:21pm)
[RUBY-SORT]	Ruby Algorithms: Sorting, Trie & Heaps http://www.igvita.com/2009/03/26/ruby-algorithms-sorting-trie-heaps/ (visited October 10, 2010, at 10:40am)
[RUBY-PROC]	Reference Documentation of Class Proc http://ruby-doc.org/ruby-1.9/classes/Proc.html (visited October 10, 2010, at 10:45am)