INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de
Electrónica e Telecomunicações e de Computadores

XAdES4j — a Java Library for XAdES Signature Services

Luís Filipe dos Santos Gonçalves

(Licenciado)

Trabalho de projecto para obtenção do grau de Mestre
em Engenharia Informática e de Computadores

Orientador

Mestre Pedro Miguel Henriques dos Santos Félix

Júri

Presidente:	Professor Adjunto Pedro Alexandre de Seia Cunha Ribeiro Pereira
Vogais:	Mestre Fernando Manuel Gomes de Sousa
	Licenciado José Luís Falcão Cascalheira
	Mestre Pedro Miguel Henriques dos Santos Félix

Dezembro de 2010

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de
Electrónica e Telecomunicações e de Computadores

XAdES4j — a Java Library for XAdES Signature Services

Trabalho de projecto para obtenção do grau de Mestre
em Engenharia Informática e de Computadores

AUTOR:

(Luís Filipe dos Santos Gonçalves)

ORIENTADOR:

(Pedro Miguel Henriques dos Santos Félix)

Abstract

Electronic communications are emerging as a way of doing business and as a basis for e-government services. In this context, electronic signatures provide an essential security foundation, ensuring not only integrity but also message and signer authentication.

The European Union has recognized the importance of electronic signatures, namely on its Directive 1999/93/EC, where a Community framework for electronic signatures is established. Following that directive, the XML Advanced Electronic Signatures specification defines format and processing rules for electronic signatures that remain valid over long periods of time and whose validity cannot be later denied (repudiated) by the signer.

This document describes the development of a Java class library that enables the production and verification of XML Advanced Electronic Signatures, extending the Java platform which only has native support for basic XML signatures. The library has a high-level programming model which abstracts the developer from signatures' syntax and processing rules. In addition, the library's design is based on service providers which represent parts of the signature operations, such as signing key/certificate selection, certificate validation and time-stamp tokens validation. The library is extensible and allows the independent configuration of such providers without altering the core mechanisms as a whole. It is also complete, as it includes implementations of all the providers.

As a conformance test, the library is used to verify a set of signatures produced by Member States of the European Union and by another implementation of the XML Advanced Electronic Signatures specification.

Keywords: digital signatures, XML, XML-DSIG, advanced signatures, XAdES, Java

Resumo

As comunicações electrónicas são cada vez mais o meio de eleição para negócios entre entidades e para as relações entre os cidadãos e o Estado (*e-government*). Esta diversidade de transacções envolve, muitas vezes, informação sensível e com possível valor legal. Neste contexto, as assinaturas electrónicas são uma importante base de confiança, fornecendo garantias de integridade e autenticação entre os intervenientes.

A produção de uma assinatura digital resulta não só no valor da assinatura propriamente dita, mas também num conjunto de informação adicional acerca da mesma, como o algoritmo de assinatura, o certificado de validação ou a hora e local de produção. Num cenário heterogéneo como o descrito anteriormente, torna-se necessária uma forma flexível e interoperável de descrever esse tipo de informação. A linguagem XML é uma forma adequada de representar uma assinatura neste contexto, não só pela sua natureza estruturada, mas principalmente por ser baseada em texto e ter suporte generalizado.

A recomendação *XML Signature Syntax and Processing* (ou apenas *XML Signature*) foi o primeiro passo na representação de assinaturas em XML. Nela são definidas sintaxe e regras de processamento para criar, representar e validar assinaturas digitais. As assinaturas XML podem ser aplicadas a qualquer tipo de conteúdos digitais identificáveis por um URI, tanto no mesmo documento XML que a assinatura, como noutra qualquer localização. Além disso, a mesma assinatura XML pode englobar vários recursos, mesmo de tipos diferentes (texto livre, imagens, XML, etc.).

À medida que as assinaturas electrónicas foram ganhando relevância tornou-se evidente que a especificação *XML Signature* não era suficiente, nomeadamente por não dar garantias de validade a longo prazo nem de não repudiação. Esta situação foi agravada pelo facto da

especificação não cumprir os requisitos da directiva 1999/93/EC da União Europeia, onde é estabelecido um quadro legal para as assinaturas electrónicas a nível comunitário.

No seguimento desta directiva da União Europeia foi desenvolvida a especificação *XML Advanced Electronic Signatures* que define formatos XML e regras de processamento para assinaturas electrónicas não repudiáveis e com validade verificável durante períodos de tempo extensos, em conformidade com a directiva. Esta especificação estende a recomendação *XML Signature*, definindo novos elementos que contêm informação adicional acerca da assinatura e dos recursos assinados (propriedades qualificadoras).

A plataforma Java inclui, desde a versão 1.6, uma API de alto nível para serviços de assinaturas digitais em XML, de acordo com a recomendação *XML Signature*. Contudo, não existe suporte para assinaturas avançadas. Com este projecto pretende-se desenvolver uma biblioteca Java para a criação e validação de assinaturas XAdES, preenchendo assim a lacuna existente na plataforma.

A biblioteca desenvolvida disponibiliza uma interface com alto nível de abstracção, não tendo o programador que lidar directamente com a estrutura XML da assinatura nem com os detalhes do conteúdo das propriedades qualificadoras. São definidos tipos que representam os principais conceitos da assinatura, nomeadamente as propriedades qualificadoras e os recursos assinados, sendo os aspectos estruturais resolvidos internamente.

Neste trabalho, a informação que compõe uma assinatura XAdES é dividia em dois grupos: o primeiro é formado por características do signatário e da assinatura, tais como a chave e as propriedades qualificadoras da assinatura. O segundo grupo é composto pelos recursos assinados e as correspondentes propriedades qualificadoras. Quando um signatário produz várias assinaturas em determinado contexto, o primeiro grupo de características será semelhante entre elas. Definiu-se o conjunto invariante de características da assinatura e do signatário como perfil de assinatura. O conceito é estendido à verificação de assinaturas englobando, neste caso, a informação a usar nesse processo, como por exemplo os certificados raiz em que o verificador confia. Numa outra perspectiva, um perfil constitui uma configuração do serviço de assinatura correspondente.

O desenho e implementação da biblioteca estão também baseados no conceito de fornecedor de serviços. Um fornecedor de serviços é uma entidade que disponibiliza determinada informação ou serviço necessários à produção e verificação de assinaturas, nomeadamente: selecção de chave/certificado de assinatura, validação de certificados, interacção com servidores de *time-stamp* e geração de XML. Em vez de depender directamente da informação em

causa, um perfil — e, consequentemente, a operação correspondente — é configurado com fornecedores de serviços que são invocados quando necessário. Para cada tipo de fornecedor de serviços é definida um interface, podendo as correspondentes implementações ser configuradas de forma independente. A biblioteca inclui implementações de todos os fornecedores de serviços, sendo algumas delas usadas for omissão na produção e verificação de assinaturas.

Uma vez que o foco do projecto é a especificação XAdES, o processamento e estrutura relativos ao formato básico são delegados internamente na biblioteca *Apache XML Security*, que disponibiliza uma implementação da recomendação *XML Signature*.

Para validar o funcionamento da biblioteca, nomeadamente em termos de interoperabilidade, procede-se, entre outros, à verificação de um conjunto de assinaturas produzidas por Estados Membros da União Europeia, bem como por outra implementação da especificação XAdES.

Palavras-chave: assinatura digital, XML, XML-DSIG, assinaturas avançadas, XAdES, Java.

Acknowledgements

I would like to thank:

My parents, for bearing my absence and for the endless background support

Sara, for all the love and understanding and for never letting me give up

Pedro Félix, for the orientation, for his experienced ideas, and for the availability throughout the project

José Simão, for the many enlightening talks and clarified issues

João Trindade, for reviewing the text with an outside perspective

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

XML eXtensible Markup Language [1].

DOM Document object model [2].

XML-DSIG XML Digital Signatures [3].

XAdES XML Advanced Electronic Signatures [4].

TSA Time-stamping authority [5].

PKI Public key infrastructure [6].

CA Certificate authority [6].

CRL Certificate revocation list [6].

OCSP Online certificate status protocol [7].

CHAPTER 1

Introduction

This document describes the project undertaken as a partial requirement for obtaining the Master's degree in Computer Engineering at Instituto Superior de Engenharia de Lisboa (ISEL). The project's goal is to develop a Java class library for XML Advanced Electronic Signatures [4] services, designated by *XAdES4j* (*XAdES for Java*).

The technological growth in the last few decades brought computer systems and digital information to the center of services and businesses. The Internet became a needful means of communication, supporting a wide range of services from simple sharing of multimedia contents to complex commercial transactions, through e-banking and services in the public sector, namely in government/citizen communications.

This diversity of transactions often encompasses sensitive information, possibly with legal value. The legal value of a document comes from the guarantee of origin, i.e. the identification of its producer; hence, as important as the confidentiality of sensitive messages or documents is ensuring their integrity, authenticity and non-repudiation. Digital signatures in conjunction with a public key infrastructure are a means for achieving those objectives: the signature binds a document to a public key which in turn is bound to an identity through a public key certificate.

1.1 Context

The output of a digital signature operation includes not only the signature itself but also additional metadata, such as the signature algorithm and the public key or certificate for signature validation or even more general information like a signature time stamp. The variety of services and actors on the Internet context requires a flexible and interoperable representation of both the signature and the additional metadata. The XML language [1] fulfills those needs: it has widespread support, namely in interoperability scenarios, not only due to its semantically rich and structured data, but also to its text-based nature.

The first step on XML signatures was taken by the World Wide Web Consortium (W3C) on its *XML Signature Syntax and Processing* [3] recommendation (or simply *XML Signatures*) which specifies XML syntax and processing rules for creating, representing and verifying digital signatures. The specification defines a basic format for signatures that cover a variable number of resources (namely XML) and takes into account some commonly used information, such as algorithms and keys.

As electronic signatures became more relevant new capabilities were needed, namely non-repudiation and long term validity. These subjects were not accounted in XML Signatures, hence highlighting the need for more advanced formats that can support the further development of electronic business and electronic government. Furthermore, the European Union recognized the importance of electronic signatures and has made efforts on establishing a legal framework to support their usage through the European Electronic Signature Directive [8]. Thus, the need for new formats is aggravated by the conformance to those legal backgrounds.

XML Advanced Electronic Signatures (XAdES) [4] is an European standard that defines XML formats for advanced electronic signatures that remain valid over long periods of time and are compliant with the European Directive. It defines XML elements that provide information on both the signature and the resources being signed (*qualifying properties*) — such as the signing certificate, time-stamps and identification of the commitments taken towards the signed resources — as well as the rules to incorporate those element into the existing XML Signatures format. Certain qualifying properties can be combined to obtain *signature forms*; if a signature has a specific form, it must include a specific set of properties. The standard defines main forms, for general usage, and extended forms, for very long term validity.

The usage of XAdES has gone beyond the European boundaries and plays an important role in the long term securing of electronic documents. An increasing number of European countries and Japanese organizations are adopting the usage of XAdES [9] in areas like e-

invoicing, digital accounting and e-government. Furthermore, the Brazilian government also defines XAdES as the accepted standard for digital signatures in Brazil [10].

XAdES is also gaining relevance on day-to-day products, namely in the Microsoft Office suite. The current version of the software (2010) enables signing text documents, spreadsheets and slide-shows with signatures ranging from basic XML Signatures to some of the more advanced XAdES forms [11].

1.2 Motivation and Goals

The Java platform has native support for XML Signatures services through a mechanism and implementation-independent API. However, it does not support XAdES signatures, which is a significant lacuna given their growing importance. The research conducted during the project also made it clear that existing open XAdES implementations for the Java platform are incomplete or have significant limitations. *XAdES4j* aims at bridging these gaps by providing a flexible and complete Java library for XAdES signatures services.

The main goal of the project is to create a library that enables the production and verification of electronic signatures in conformance to XAdES 1.4.1. The XAdES specification defines four main signature forms and three extended forms, each one requiring the presence of certain qualifying properties. To be compliant with XAdES, implementors must support at least one of the main forms. *XAdES4j* aims to support the production and verification of signatures in all the main XAdES forms, including all the optional qualifying properties — i.e. properties other than the ones required by the signature forms. Moreover, the library enables the enrichment of an existing signature by adding extra qualifying properties. Optionally, the library may support the production and/or verification of signatures according to the extended forms.

The XAdES specification defines the information that makes up an advanced signature as well as its structure. Gathering the necessary data to produce or verify a signature might be as hard as composing its final structure and, more important, it is a process that depends on the usage scenarios. Selecting signing keys, validating certificates, interacting with time-stamp providers and selecting signature properties are all tasks that can be accomplished in several ways, although they do not change the signature production and verification processes as whole. The library's design should take these aspects into account, being flexible enough to accommodate the different use-cases.

Finally, as a conformity test the library must successfully verify a set of signatures pro-

duced by Member States of the European Union.

1.3 Solution Overview

XAdES4j enables the production and verification of XAdES signatures in any of the main forms defined by the specification. During verification, a signature's form may also be enriched up to the extended forms. The library builds on the Apache XML Security API [12] for basic XML signatures but without directly exposing the existing types. Instead, new types are created to represent the main concepts of XML Signatures and XAdES, namely data objects and qualifying properties, as illustrated in Figure 1.1. These types do not reflect the corresponding XML structure as the details on signatures structure, forms and processing rules are resolved internally.

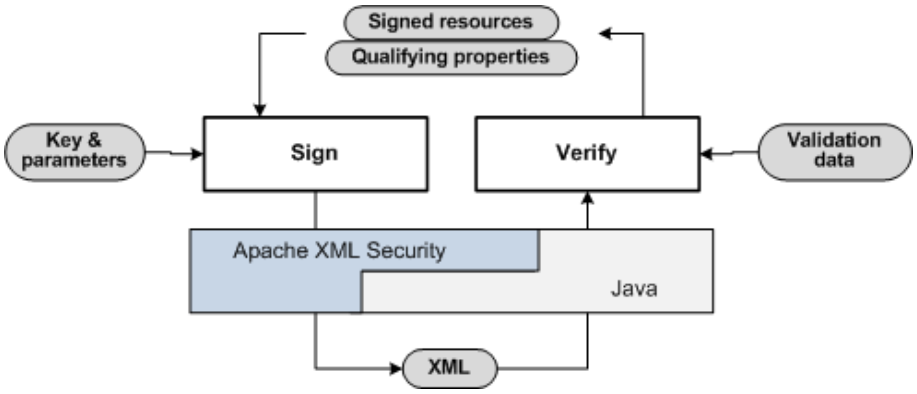

Figure 1.1: Solution overview

This approach includes abstracting the developer from creating the actual content of the qualifying properties. For instance, some properties involve calculating digests of certificates or policy documents and others include determining inputs for time-stamps. To that end, additional services are needed, such as obtaining message digest engines and interacting with TSAs. Furthermore, other services are required for signature production and verification, namely signing certificate selection, certificate validation and time-stamp tokens validation. *XAdES4j* handles these services as exchangeable components, which are plugged into the library through well defined interfaces and can be configured independently without altering the core mechanisms. Also, the library is complete, meaning it includes implementations for the different services, some of them used by default.

The work is based on the following versions of the different components:

- Java Runtime Environment: build 1.6.0_21-b07 (Java 6, update 21).

- JAXB: reference implementation in JDK 6, version 2.1.10.
- Apache XML Security: version 1.4.3.
- Google Guice: version 2.0.

All the development and tests were done using the NetBeans IDE, version 6.7.1, on a Windows 7 32-bit platform.

1.4 Document Organization

1.4.1 Structure

The document consists of six chapters and four appendixes. The following is a brief description of the remaining parts.

Chapter 2 — Normative Basis presents the XML-DSIG and XAdES specifications, the foundations for this project. The XAdES specification is presented with reasonable detail, as it is the focus of the implementation.

Chapter 3 — State of the Art contains a brief analysis of existing solutions for both XML-DSIG and XAdES signature services.

Chapter 4 — Architecture describes the library's core concepts and how the library's programming model and architecture are defined around them.

Chapter 5 — Implementation documents the library's implementation details.

Chapter 6 — Conclusions presents the concluding remarks, including a brief comparison with other solutions as well as an analysis of the project's limitations and possible enhancements.

Appendix A — Package Organization provides a brief description of the library's logical organization.

Appendix B — Dependency Injection with Guice contains a brief introduction to *Guice*, a dependency injection framework used in the library.

Appendix C — Exception Model presents the exception hierarchy resulting from the adopted exception model.

Appendix D — Test Cases provides brief descriptions of the test cases implemented to validate the library's features.

1.4.2 Conventions and Nomenclature

Throughout the text the names of XML elements are written using a `type writer` font, while names of Java types and corresponding members are presented using a sans serif font. In addition, whenever a new concept in introduced the *italic shape* is used.

The code snippets along the text are also presented with a `sans serif` font and do not include exception declarations (throws clauses) on method definitions.

XAdES4j, i.e., the deliverable resulting from the current project, is frequently referred to as *the library*.

CHAPTER 2

Normative Basis

The *XML Advanced Electronic Signatures* (XAdES) specification [4], which is the focus of this project, builds on the earlier *XML Digital Signatures* (XML-DSIG) [3]. The later specifies XML syntax and processing rules for representing and processing digital signatures, while XAdES defines a set of XML elements which contain information that further qualifies a signature. The current chapter establishes the normative basis that is needed for the remainder of this document: Section 2.1 presents the XML-DSIG syntax and rules; Section 2.2 describes XAdES, starting by its legal background and proceeding to its main elements and signature formats.

2.1 XML Digital Signatures

XML Signature [3] has been an official W3C Recommendation since February 2002 [13] and specifies XML syntax and processing rules for creating, representing and verifying digital signatures, including commonly used information such as algorithms and keys. Furthermore, the syntax supports extending the meaning of the signature with application specific semantics that can also be signed. The signatures provide integrity, message authentication and signer authentication for data of any type, whether located within the XML document that includes the signature or elsewhere. Although supporting any digital content, XML signatures have some focus on both accounting for and taking advantage of the nature of XML data. One important aspect is the possibility of signing portions of documents. For example,

in a scenario where a document flows between persons, each participant may wish to sign only the portion they are responsible for. With XML and surrounding technologies, one can easily refer to portions of documents. Furthermore, XML has a widespread support and plays a major role on today's systems, not only due to its structured and semantically rich data, but also to its text-based, web-ready nature. On this scenario, the security aspects also have to be addressed using XML, hence the need of standards as *XML Signature*.

2.1.1 Principles

XML signatures apply to any type of digital content, designated by *data objects*, addressable by an URI, either within the same document as the signature — via fragment identifiers — or on external resources (accessible through the network, for example). Moreover, an XML signature can sign multiple resources, even from different types. For example, a single signature might cover character-encoded data (HTML), binary-encoded data (a JPG), XML-encoded data and a specific section of an XML file.

The generation of XML signatures over data objects has two steps, as illustrated in Figure 2.1. First, each data object is subject to a digest algorithm. The set of resulting digests is placed in an XML element along with other information and then that element is digested and cryptographically signed.

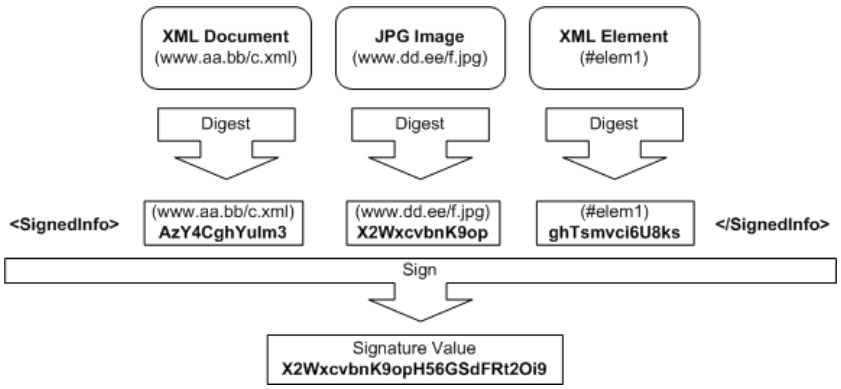

Figure 2.1: Main steps of signature generation

The signature value and information about the data objects being signed are registered within the `Signature` element, which represents an XML digital signature according to the schema fragment in Listing 2.1.

The `SignatureValue` element contains the actual value of the digital signature, encoded in base64. The `KeyInfo` and `Object` elements contain information related to the key used to generate the signature and any additional data, respectively. None of these elements

Listing 2.1: Signature element

```
<element name="Signature" type="ds:SignatureType"/>
<complexType name="SignatureType">
  <sequence>
    <element ref="ds:SignedInfo"/>
    <element ref="ds:SignatureValue"/>
    <element ref="ds:KeyInfo" minOccurs="0"/>
    <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

is directly signed. On the other hand, the `SignedInfo` element is the information that is actually signed: it contains the signature algorithm and the references to the data objects being signed, as shown in Listing 2.2. Each reference contains the object identification (URI) and its digest, as result of the first step of the signature generation.

Listing 2.2: SignedInfo element

```
<element name="SignedInfo" type="ds:SignedInfoType"/>
<complexType name="SignedInfoType">
  <sequence>
    <element ref="ds:CanonicalizationMethod"/>
    <element ref="ds:SignatureMethod"/>
    <element ref="ds:Reference" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

The signature algorithm is included within this element to prevent attacks based on replacing it with one that is weaker. Since the algorithm's name is signed any change would fail the signature validation. To promote interoperability, implementations of XML Signatures must provide the *DSA-SHA1* signature algorithm and are recommended to also provide *RSA-SHA1*.

The `SignedInfo` element does not include any signature or digest properties (such as signature time stamp) nor other application specific data. Such information may be included within `Object` elements, which can be accounted to the signature, if necessary, with a *same-*

document reference from within `SignedInfo`.

XML signatures have different designations depending on the location of the data objects being signed, as illustrated on Figure 2.2. When the `Signature` element is a child of the XML content being signed, the signature is an *enveloped signature*. In this case the `Signature` element must be excluded from the calculation of the signature value because its contents changes when the signature value is appended. On the other hand, if the signature is applied over the content of an `Object` element of the signature itself, it is an *enveloping signature*. The object or its content is identified through a `Reference` element within `SignedInfo`. When neither of the previous applies, the signature is a *detached signature*. The signed resource can be present within the same document as the `Signature` element or it can be external to the XML document. An example of a detached signature is the inclusion of a signature on a SOAP header as part of *WS-Security* [14].

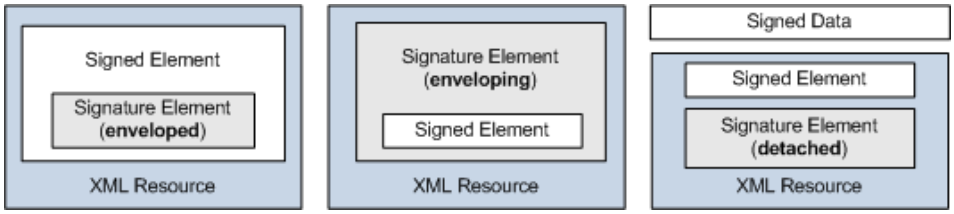

Figure 2.2: Different types of signatures

2.1.2 Data Object References

The identification of signed data objects is done via the `Reference` elements within `SignedInfo`. As imposed by the element's schema shown in Listing 2.3, each reference includes the digest method and resulting digest value calculated over the identified data object. The data object is identified via the `URI` attribute which can be omitted on at most one `Reference` in a `SignedInfo`. In this case, the receiving application is expected to know how to identify the target object. Also, signature applications are recommended to be able to dereference URIs in the HTTP scheme.

The `URI` attribute may also contain a *same-document reference* which consists of a number sign (#) followed by a fragment that conforms to the *XPointer*[15] syntax. Dereferencing such a reference is done by evaluating the XPointer with respect to the XML document containing the `URI` attribute and must result in an XPath node-set. When a null URI (`URI=""`) is used the node-set includes every non-comment node of the document. On the other hand, the result of dereferencing an absolute or relative URI must be an octet stream.

Listing 2.3: Reference element

```
<element name="Reference" type="ds:ReferenceType"/>
<complexType name="ReferenceType">
  <sequence>
    <element ref="ds:Transforms" minOccurs="0"/>
    <element ref="ds:DigestMethod"/>
    <element ref="ds:DigestValue"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
  <attribute name="URI" type="anyURI" use="optional"/>
  <attribute name="Type" type="anyURI" use="optional"/>
</complexType>
```

Data objects may be subject to a set of transformations prior to digesting, which are represented by the `Transforms` element (Listing 2.4). These transformations describe how the signer obtained the data object in the form it was digested. The signature verifier may obtain the digested content with another method, in particular from a different URI, such as a local storage. If no transformations are specified, the data object's content is digested directly.

`Transforms` is an ordered list of processing steps with the input of the first `Transform` being the result of dereferencing the URI of the `Reference`. The outputs and inputs are chained until the last `Transform`, whose output is passed to the digest algorithm. The data-type of the result of URI dereferencing and subsequent transformations is either an octet stream or a XPath node-set. Whenever a data object is an octet stream and the next transformation requires a node-set, signature applications must attempt to parse the octets. Likewise, if the data object is a node-set and the next transform requires octets, signature applications must attempt to convert the node-set using Canonical XML. This rule is also applied between URI dereferencing and content digest.

A `Transform` consists of an algorithm name and its parameters (if any) and can include operations such as canonicalization, XSLT, XPath filtering and XML Schema validation. XPath transformations allow the signer to select the portions of the document he wants to sign. This is one of the advantages of XML Signatures, being reinforced by the `XPath` element on the `Transform`'s schema. The definition of the `Transform` element also allows application-specific algorithms. However, if interoperability is a concern, applications should adhere to the list of standard transformations.

Listing 2.4: Transforms element

```
<element name="Transforms" type="ds:TransformsType"/>
<complexType name="TransformsType">
  <sequence>
    <element ref="ds:Transform" maxOccurs="unbounded"/>
  </sequence>
</complexType>

<element name="Transform" type="ds:TransformType"/>
<complexType name="TransformType" mixed="true">
  <choice minOccurs="0" maxOccurs="unbounded">
    <any namespace="##other" processContents="lax"/>
    <!-- (1,1) elements from (0,unbounded) namespaces -->
    <element name="XPath" type="string"/>
  </choice>
  <attribute name="Algorithm" type="anyURI" use="required"/>
</complexType>
```

2.1.3 Signature Generation and Validation

The generation of XML Signatures has two required steps. The first step is *reference generation* and consists on generating the `Reference` elements, namely the digest values. For each data object being signed the following steps have to be executed:

1. Apply the specified transformations to the data object, if any;
2. Calculate the digest value over the resulting node-set or octet-stream;
3. Create a `Reference` element with object identification, transformations, the digest algorithm, and the digest value.

The second step is *signature generation* and consists on generating the `SignatureValue` over `SignedInfo`. The following steps are executed:

1. Create the `SignedInfo` element with the `SignatureMethod`, the `CanonicalizationMethod` and the `Reference`s generated in the first step;
2. Canonicalize `SignedInfo` and then calculate the `SignatureValue` over it;
3. Create the `Signature` element with `SignedInfo` and `SignatureValue` elements, and optionally the `Object` and `KeyInfo` elements.

Canonicalization is the generation of an octet stream representation of an XML document after performing a series of steps recommended by the W3C specifications Canonical XML [16] and Exclusive XML Canonicalization [17]. This physical representation of the XML data is used to determine whether two XML documents are identical. Even a slight variation in white spaces will result in a different hash for an XML document. Thus, to guarantee that logically-identical `SignedInfo` elements give identical digital signatures, an XML canonicalization transform is employed.

The validation of the signatures is also done in two steps: *reference validation* and *signature validation*. In reference validation the digest value of each data object is validated by the following actions:

1. Obtain the data object either from dereferencing the `Reference`'s URI and applying transformations or from another place, such as a local storage;
2. Digest the object using the algorithm specified in the `Reference`;
3. Compare the resulting digest value with the one in the `Reference` and fail the validation if there is a mismatch.

In the second step the signature value over `SignedInfo` is validated. This consists of the following steps:

1. Get the keying information either from `KeyInfo` or from an external source;
2. Canonicalize the `SignedInfo` element;
3. Verify the signature value over the resulting octet-stream using the `SignatureMethod` and the keying information above.

Note that there may be valid signatures that some applications can't validate because they lack implementations for optional parts of the specification or, for instance, they are unable to dereference some URIs.

2.1.4 Keying Information

An XML signature may include information about the key to be used in signature validation by using the `KeyInfo` optional element. If `KeyInfo` is omitted, the recipient is expected to know how to identify the key based on application context. Otherwise, the `KeyInfo` element may contain keys, key names, key locations, certificates, key agreement information or

application specific data. XML Signatures defines a few types to specify keying information, where the `X509Data` element is one of the most relevant. This element contains one or more identifiers of keys or certificates according to the X.509 standard. As illustrated in Listing 2.5, the `X509Data` element may contain different information, such as a X.509 subject distinguished name, a X.509 V.3 *SubjectKeyIdentifier* extension or a X.509 certificate. At least one of the elements has to be specified in order to identify the needed certificate. Nevertheless, different elements can appear together in the same `X509Data` as long as they are related to the same certificate.

Listing 2.5: X509Data element

```
<element name="X509Data" type="ds:X509DataType"/>
<complexType name="X509DataType">
  <sequence maxOccurs="unbounded">
    <choice>
      <element name="X509IssuerSerial" type="ds:X509IssuerSerialType"/>
      <element name="X509SKI" type="base64Binary"/>
      <element name="X509SubjectName" type="string"/>
      <element name="X509Certificate" type="base64Binary"/>
      <element name="X509CRL" type="base64Binary"/>
      <any namespace="##other" processContents="lax"/>
    </choice>
  </sequence>
</complexType>
```

When multiple certificates appear in a `X509Data` element, one of them must contain the public key that validates the signature; all the others must be part of the certification chain. Furthermore, all the other elements must relate to the certificate containing the validation key.

2.1.5 Generic Data

When generating a signature additional information may be produced such as some metadata for the "true" content being signed. To account for this, XML Signatures defines a generic data placeholder: the `Object` element. This element can occur multiple times within a signature and includes attributes to specify its content's MIME type and encoding. The `Object` element is often used to place information that needs to be signed, which means it is usually referenced from `SignedInfo`. In fact, with *enveloping signatures* the content being

signed is placed within `Object` elements.

2.2 XML Advanced Electronic Signatures

Since 1997, the European Commission has made efforts on ensuring trust and security in electronic communications, towards a Community framework for digital signatures and encryption. The Directive 1999/93/EC [8] of the European Parliament establishes a legal framework for electronic signatures, acknowledging their importance to electronic communications and increasing confidence in the new technologies. The document clearly reveals the concerns on establishing a common and comprehensive solution:

> "Electronic communications and commerce necessitate electronic signatures and related services allowing data authentication: divergent rules with respect to legal recognition of electronic signatures and the accreditation of certification-service providers (...) may create a significant barrier to the use of electronic communications."
>
> "Rapid technological development and the global character of the Internet necessitate an approach which is open to various technologies and services capable of authenticating data electronically".

The directive defines an *advanced electronic signature* as an electronic signature which meets the following requirements:

1. it is uniquely linked to the signatory, by the signature value itself;
2. identifies the signatory, matching him with a name;
3. it is created using means that the signatory maintains under his control;
4. enables the detection of subsequent changes to the signed data.

The directive also specifies the requirements for *qualified certificates* and for the *certification service providers* that create them. An advanced electronic signature based on a qualified certificate and created by a *secure-signature-creation device* is named *qualified electronic signature*. The member states shall ensure that this signatures are admissible as evidence in legal proceedings. In addition, they shall ensure that, by issuing a qualified certificate, the certification service providers are liable for damage resulting of actions based on that certificate, namely due to failure on registering revocations.

Electronic signatures are expected to be used in the public sector within national and Community administrations, namely in electronic transactions with the citizens. The Portuguese citizen card (*Cartão de Cidadão*) reflects this Directive, allowing the inclusion of a qualified certificate for electronic signature operations. In fact, by March 2006 all the 25 Member States had implemented the general principles of the Directive [18]: the legal recognition of electronic signatures has been met by the transposition of the Directive into the legislation of each State. However, although many Member States and several other European countries had launched e-government applications or were planning to do so, the take up on using advanced or qualified electronic signatures was slow. Ensuing studies concluded that there were problems of mutual recognition and interoperability at a general level mostly due to the multiplicity of standardization deliverables and the lack of usage guidelines. As a result, in December 2009 the European Commission issued a standardization mandate on electronic signatures [19] to update the existing European electronic signatures standardization deliverables in order to create a rationalized framework.

2.2.1 Overview

Following the European Directive, the European Committee for Standardization (CEN) and European Telecommunications Standards Institute (ETSI) have developed a number of standards regarding both hardware and software electronic signature solutions, namely advanced signature formats. One of those specifications is *XML Advanced Electronic Signatures* (XAdES) [4] which defines XML formats for advanced electronic signatures that remain valid over long periods (even if repudiation is attempted) and are compliant with the European Directive by incorporating additional qualifying information.

The qualifying information (*properties*) that has to be added to an electronic signature in order to satisfy the aforementioned requirements was previously identified in *CMS Advanced Electronic Signatures* (CAdES) [20]. XAdES specifies XML schema definitions for new elements that carry similar properties and are used to amend XML-DSIG [3] signatures. It also specifies how those elements should be incorporated into XML-DSIG, namely by adding one new `ds:Object`[1] element containing the qualifying properties.

The qualifying properties can be grouped according to their information, namely:

- *Signature policy identifier* — unambiguously identifies the signature policy under which the signature has been produced. The policy clarifies the commitments that the signer

[1] The *ds* prefix is used throughout the chapter to represent the XML-DSIG namespace

intends to assume towards the signed data objects.

- Validation data properties — incorporate all the validation material into the signature. This includes certificate chains, certificate revocation lists (CRL) [6] and on-line certificate status protocol (OCSP) [7] responses. XAdES supports both validation data and references to them (identifiers).

- Time-stamp token properties — contain time-stamp tokens covering different parts of the signature, such as elements defined in XML-DSIG and validation data properties. These properties are the base for long term validity as they are proof that the signature was generated before a given time, such as the time of a certificate revocation. Section 2.2.3 details time-stamping in XAdES.

- Other properties used in a wide range of scenarios, such as the signing certificate and the signature production location.

Regardless of their content, there are two main types of qualifying properties: *signed properties* and *unsigned properties*. The first ones are secured by the signature, meaning they have a corresponding `ds:Reference` within `ds:SignedInfo`. This also implies that the signer detains all that information when generating the signature. On the other hand, unsigned properties are not secured by the signature and can be added by the signer, the verifier or other parties after the production of the signature. These properties contain information that is secured by other means, such as certificates and CRLs. The qualifying properties can also be classified according to their target: some properties apply to the signature (or the signer) while others apply to the data objects being signed. Classifying the properties along these two axes results in four specific property types, as illustrated in Figure 2.3.

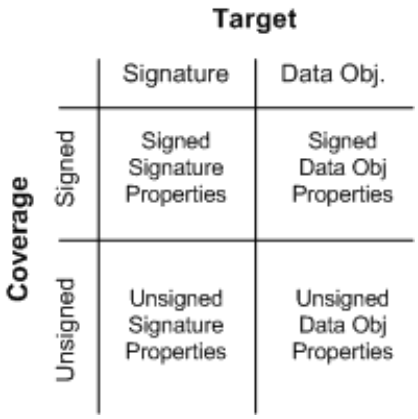

Figure 2.3: Classifying the qualifying properties

2.2.2 The QualifyingProperties Element

All the qualifying properties that should be added to an XML signature are contained in the `QualifyingProperties` element, which has the structure shown in Listing 2.6.

Listing 2.6: QualifyingProperties element

```
<xsd:element name="QualifyingProperties" type="QualifyingPropertiesType"/>
<xsd:complexType name="QualifyingPropertiesType">
  <xsd:sequence>
    <xsd:element name="SignedProperties" type="SignedPropertiesType"
        minOccurs="0"/>
    <xsd:element name="UnsignedProperties" type="UnsignedPropertiesType"
        minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="Target" type="xsd:anyURI" use="required"/>
  <xsd:attribute name="Id" type="xsd:ID" use="optional"/>
</xsd:complexType>
```

The `SignedProperties` element is the container of signed qualifying properties, while the `UnsignedProperties` element contains the unsigned qualifying properties. The `Target` attribute is mandatory and must refer to the `Id` attribute of the target `ds:Signature`. Both types of properties may qualify the signature and/or the signer or the signed data objects, as illustrated in Listing 2.7 for the `SignedProperties` element.

Listing 2.7: SignedProperties element

```
<xsd:complexType name="SignedPropertiesType">
  <xsd:sequence>
    <xsd:element name="SignedSignatureProperties"
        type="SignedSignaturePropertiesType" minOccurs="0"/>
    <xsd:element name="SignedDataObjectProperties"
        type="SignedDataObjectPropertiesType" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="Id" type="xsd:ID" use="optional"/>
</xsd:complexType>
```

All the properties in `SignedDataObjectProperties` qualify the data objects after the required transformations have been applied.

Incorporating Qualifying Properties into the Signature

The qualifying properties are incorporated into XML-DSIG signatures with a new `ds:Object` element in one of two ways:

1. Direct incorporation — the `QualifyingProperties` element is as a child of `ds:Object`.

2. Indirect incorporation — one or more `QualifyingPropertiesReference` elements appear as children of the `ds:Object` element.

The `QualifyingPropertiesReference` element contains a URI that refers a `QualifyingProperties` element that is stored elsewhere. Direct incorporation is more straightforward and should be the most common scenario.

All the qualifying properties must occur within a single `ds:Object` element which may contain at most one `QualifyingProperties` element. Furthermore, all signed properties must occur within a single `QualifyingProperties`, either with direct or indirect incorporation. In order to protect this properties with the signature, the `SignedProperties` element must be covered by a `ds:Reference` of the XML signature. Additionally, the `Type` attribute of that `ds:Reference` should be set to `http://uri.etsi.org/01903#SignedProperties`. This helps verifying applications to detect the signed properties of a certain signature. Figure 2.4 illustrates the structure of a XAdES signature using direct incorporation.

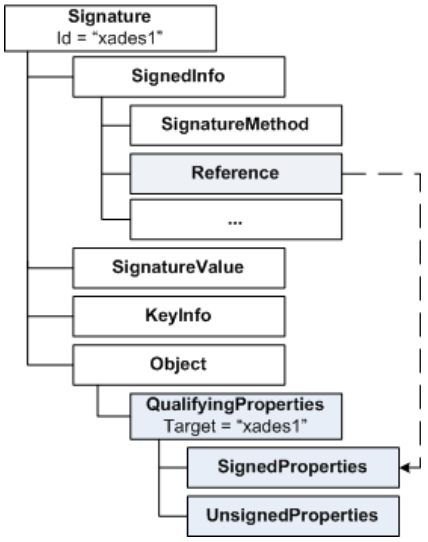

Figure 2.4: XAdES signature structure

2.2.3 Time-stamping

XAdES defines several properties that contain time-stamp tokens computed on both XML-DSIG elements and XAdES qualifying properties. A time-stamp token is obtained by sending the digest value of the given data to a Time-Stamp Authority (TSA). The returned time-stamp token is a signed data that contains the digest value, the identity of the TSA, and the time of stamping. This *trusted time* provides the first and essential step towards long term validity since it proves that the given data existed before the time of stamping. Consequently, it can be used to assure that a signature was created before the revocation of a certain certificate on the certification path or of the signing certificate itself.

The time-stamp containers defined by XAdES include:

- `SignatureTimeStamp`: a property for a time-stamp token over the signature value to protect against repudiation in case of a key compromise.
- Properties that contain time-stamp tokens proving that some or all the data objects to be signed have been created before some time: `IndividualDataObjectsTimeStamp` and `AllDataObjectsTimeStamp`.
- `RefsOnlyTimeStamp`: contains a time-stamp token over all certificate and revocation information references.

XAdES defines the `XAdESTimeStampType` for containing time-stamp tokens computed on data objects of the XAdES signature. The type's schema is presented in Listing 2.8. The base type exists because another container type is defined for time-stamps over external objects, even though it is not actually used.

There are two mechanisms for identifying data objects covered by the time-stamp tokens:

1. Explicit — uses the `Include` element for referencing specific data objects and for indicating their contribution to the input of the digest computation;
2. Implicit — certain time-stamp container properties implicitly define the data objects that are covered by the time-stamp and how they contribute to the input of the digest computation. This is the case of the `SignatureTimeStamp` property, which always applies to the `ds:SignatureValue` element.

When the explicit mechanism is used, `Include` elements identify data objects that are time-stamped and the order of appearance indicates how the data objects contribute in

Listing 2.8: Time-stamp container type

```
<xsd:complexType name="IncludeType">
  <xsd:attribute name="URI" type="xsd:anyURI" use="required"/>
  <xsd:attribute name="referencedData" type="xsd:boolean" use="optional"/>
</xsd:complexType>

<xsd:complexType name="XAdESTimeStampType">
  <xsd:complexContent>
    <xsd:restriction base="GenericTimeStampType">
      <xsd:sequence>
        <xsd:element ref="Include" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="ds:CanonicalizationMethod" minOccurs="0"/>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="EncapsulatedTimeStamp"
              type="EncapsulatedPKIDataType"/>
          <xsd:element name="XMLTimeStamp" type="AnyType"/>
        </xsd:choice>
      </xsd:sequence>
      <xsd:attribute name="Id" type="xsd:ID" use="optional"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

the generation of the input to the digest computation. If the `Include` element covers a `ds:Reference` the attribute `referencedData` may be present. If its value is set to *true*, the time-stamp is computed on the result of processing the corresponding `ds:Reference` according to the XML-DSIG processing model. If the attribute is not present or its value is *false*, the time-stamp is computed on the `ds:Reference` element itself. On both cases, if the data resulting from the `Include` element is a XML node set, it is canonicalized with the algorithm indicated by the `ds:CanonicalizationMethod` element, or the standard specified by XML-DSIG if the element is omitted.

The time-stamp token generated by the TSA can be either an ASN.1 data object as defined in [5], using the `EncapsulatedTimeStamp` element, or it can be encoded as XML, using the `XMLTimeStamp` element.

2.2.4 Main Signature Formats

The XAdES specification defines four main signature forms: the *Basic Electronic Signature* (XAdES-BES), the *Explicit Policy based Electronic Signature* (XAdES-EPES), the *Electronic Signature with Time* (XAdES-T) and the *Electronic Signature with Complete Validation Data References* (XAdES-C), which are obtained by combination of the various qualifying properties. A XAdES signature is one that is created with one of these formats.

Basic Electronic Signature

The XAdES-BES is the minimum form for an electronic signature that satisfies the legal requirements defined in the European Directive. This form does not provide enough information for the signature to be verified in the long term.

The BES form mandates the protection of the signing certificate with the signature in order to prevent its simple substitution. If the certificate is simply added after the signed data, it is subject to various substitution attacks. For example, a certificate could be issued to someone with the public key of someone else. If a signing certificate is not protected, it could be substituted by the forged certificate and the message would appear to be signed by the wrong person.

The signing certificate can be protected in one of two ways: the first is the inclusion of the `SigningCertificate` signed property; the other is including the certificate in a `ds:X509Data` within the `ds:KeyInfo` and having a `ds:Reference` referencing `ds:KeyInfo`, built in such a way that at least the signing certificate is actually signed. The `SigningCertificate` property contains references to certificates and digest values computed on them. The certificate used to verify the signature has to be identified in the sequence, but the signature policy may mandate other certificates of the certificate chain to be present. The schema for the `SigningCertificate` element is shown in Listing 2.9.

Each `IssuerSerial` element contains the identifier of one certificate. If the signature contains a `ds:X509IssuerSerial` element within `ds:KeyInfo` its value must be consistent with the corresponding element in the property. Note that XML-DSIG mandates that the `ds:X509IssuerSerial` refers to the certificate containing the validation key. The digests over the certificates are base64-encoded calculated over the DER-encoded certificate and allow the detection of any changes on their structure since they are checked as part of the validation process.

Once the verifier has gotten the signing certificate it should check it against the references

Listing 2.9: SigningCertificate element

```
<xsd:element name="SigningCertificate" type="CertIDListType"/>
<xsd:complexType name="CertIDListType">
  <xsd:sequence>
    <xsd:element name="Cert" type="CertIDType" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="CertIDType">
  <xsd:sequence>
    <xsd:element name="CertDigest" type="DigestAlgAndValueType"/>
    <xsd:element name="IssuerSerial" type="ds:X509IssuerSerialType"/>
  </xsd:sequence>
  <xsd:attribute name="URI" type="xsd:anyURI" use="optional"/>
</xsd:complexType>
<xsd:complexType name="DigestAlgAndValueType">
  <xsd:sequence>
    <xsd:element ref="ds:DigestMethod"/>
    <xsd:element ref="ds:DigestValue"/>
  </xsd:sequence>
</xsd:complexType>
```

in the `SigningCertificate` property. The first step is to find the reference with the same issuer name and serial number. After that, check that the issuer and serial number are equal to the ones in `ds:X509IssuerSerial`, if present. Finally, check if the digest value is correct. If the property contains references to other certificates in the certification path, the verifier should also check them using the same procedure.

If the `SigningCertificate` property is not used and the certificate is within `ds:KeyInfo`, there is no verification other than the one defined in XML-DSIG.

In addition to the `SigningCertificate` property, a XAdES-BES signature may also contain other properties:

- `SigningTime` — the time at which the signer purportedly performed the signing process.
- `DataObjectFormat` — provides information that describes the format of a signed data object. It should be used when the signed data is to be presented to humans and the presentation format is not implicit within the data.
- `CommitmentTypeIndication` — the type of commitment made by the signer towards

some or all the signed data objects.

- `SignerRole` — the position of the signed within a company or organization. In many cases, the signer's identity itself is not that important, but being sure of the signer's role is fundamental.

- `SignatureProductionPlace` — the purported place where the signature creation took place.

- `IndividualDataObjectsTimeStamp` and `AllDataObjectsTimeStamp` — time-stamps computed over some or all signed data objects before the signature creation.

- `CounterSignature` — contains one countersignature (XML-DSIG or XAdES) of the qualified signature. The `ds:SignedInfo` element of the countersignature must contain a reference to the `ds:SignatureValue` of the embedding XAdES signature.

All these properties are optional and may be used on the other signature forms since they build on XAdES-BES.

Explicit Policy Electronic Signature

XAdES-EPES builds up on XAdES-BES by incorporating the `SignaturePolicyIdentifier` signed property which indicates that a signature policy must be used for signature validation.

A signature policy is a set of rules for the creation and validation of an electronic signature, under which the signature can be deemed valid. Moreover, the policy establishes the commitments of the signer towards the signed data objects. If no signature policy is identified then the signature may be assumed to have been generated/verified without any policy constraints.

The signature policy needs to be available in human readable form so that it can be assessed to meet the requirements of the legal and contractual context in which it is being applied. Nevertheless, to facilitate the automatic processing of an electronic signature, the parts of the policy which specify the electronic rules for the creation and validation of the signature also need to be in a computer processable form.

The `SignaturePolicyIdentifier` property is a way to identify the signature in use. The actual policies are not specified by XAdES but ETSI also defines a XML format for signature policies [21] that may be automatically processed. The schema for the `SignaturePolicyIdentifier` element is shown in Listing 2.10.

Listing 2.10: SignaturePolicyIdentifier element

```
<xsd:element name="SignaturePolicyIdentifier"
    type="SignaturePolicyIdentifierType"/>

<xsd:complexType name="SignaturePolicyIdentifierType">
  <xsd:choice>
    <xsd:element name="SignaturePolicyId" type="SignaturePolicyIdType"/>
    <xsd:element name="SignaturePolicyImplied"/>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="SignaturePolicyIdType">
  <xsd:sequence>
    <xsd:element name="SigPolicyId" type="ObjectIdentifierType"/>
    <xsd:element ref="ds:Transforms" minOccurs="0"/>
    <xsd:element name="SigPolicyHash" type="DigestAlgAndValueType"/>
    <xsd:element name="SigPolicyQualifiers"
        type="SigPolicyQualifiersListType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

XAdES defines two ways of identifying the signature policy: explicit and implied. In the first one, the `SignaturePolicyId` element is present, which contains an explicit and unambiguous identifier of the policy. It also contains the hash value of the signature policy document, so it can be verified that the policy selected by the signer is the one being used by the verifier. The digest value is calculated after a series of transformations, processed as defined by XML-DSIG. In the second case, the `SignaturePolicyImplied` element is present, indicating that the data objects being signed and other external data, such as laws or private agreements, imply the signature policy.

When the policy is not implied, the verifier should retrieve the policy document as identified by the `SigPolicyId` and apply the specified transformations. Then, it computes the digest of the output using the specified algorithm and checks its value against the one in the property. If the policy is implied, the verifier should perform any checks mandated by that policy.

Even though XAdES-EPES only specifies the `SignaturePolicyIdentifier` property, other properties may be required by the mandated policy, namely properties that contain

information to be used in the signature validation.

Electronic Signature with Time

In order to accomplish long term validity, additional data is needed to validate the electronic signatures. This *validation data* includes public key certificates, revocation information and trusted time-stamps, and may be collected by the signer and/or the verifier.

If a signer's key is revoked, her signatures should not be considered invalid if they were created before the revocation. Thus, there is the need to be able to demonstrate that the signature key was valid when the signature was created. Time-stamping an electronic signature before the revocation of the signer's key and before the expiration of the certificate provides evidence that the signature has been created while the certificate could be used.

XAdES-T is a form for which there exists a trusted time associated to the signature. The trusted time can be provided either as a time-mark or as a time-stamp token inside an unsigned property. Both bind a set of data to a particular time; the difference is that a time-mark is kept in an audit trail from its provider while a time-stamp token is returned to the signer. When a time-mark is used, no property is added because the evidence is provided by a trusted service provider on demand. On the other hand, when the trusted time is added as a property the `SignatureTimeStamp` element is used to contain a time-stamp token over the signature value. A XAdES-T signature may contain several `SignatureTimeStamp` elements, resulting from different time-stamping authorities. This property uses the implicit mechanism as the time-stamped data object is always the `ds:SignatureValue` element. The input to the digest computation is the result of the canonicalization of the `ds:SignatureValue` element and its contents.

To validate this property, the verifier first takes the `ds:SignatureValue` element and applies the canonicalization algorithm specified in the property. Then, for each time-stamp token in the property, he verifies its signature and checks if the digest present in the token has the same value as the digest computed over the canonicalized `ds:SignatureValue`. Finally, the verifier has to check time coherence to other time-stamps in the signature. For instance, the signature time-stamp has to be posterior to all the time-stamps over signed data objects, if present.

Note that signatures may be generated off line and time-stamped at a later time by anyone. The sooner the time-stamp is obtained, the better, not only to prove that the signature is previous to any revocation, but also because signature policies may specify a maximum time

difference between the signature generation (`SigningTime`) and the time-stamp.

Electronic Signature with Complete Validation Data References

When dealing with long term electronic signatures all the data used in the verification process must be stored and conveniently time-stamped for arbitration purposes. In some environments, it is convenient to archive these data within the signature. On the other hand, there may be situations where the validation data is stored outside of the signature to prevent redundant storage and reduce the signature's size. In such cases the signature must incorporate references to all validation data.

This form builds on XAdES-T by incorporating references to the full set of CA certificates in the certification path and references to revocation data used in the validation of the signer and CA certificates. Furthermore, if attribute certificates are used in the signature, XAdES-C also incorporates the corresponding references.

References to the CA certificates are held in the `CompleteCertificateRefs` optional unsigned property. The element's schema is shown in Listing 2.11.

Listing 2.11: CompleteCertificateRefs element

```
<xsd:element name="CompleteCertificateRefs"
    type="CompleteCertificateRefsType"/>
<xsd:complexType name="CompleteCertificateRefsType">
  <xsd:sequence>
    <xsd:element name="CertRefs" type="CertIDListType" />
  </xsd:sequence>
  <xsd:attribute name="Id" type="xsd:ID" use="optional"/>
</xsd:complexType>
```

The `CertIDListType` is the same as in XAdES-BES, presented in Section 2.2.4, incorporating the digest of each certificate and the issuer and serial number identifier. To verify this property, the verifier has to gain access to all the CA certificates in the certification path from the `CertificateValues` property — which, if present, contains all the certificates used to verify the signature —, from within `ds:KeyInfo` or from any other external source. Then, it must check that the property contains a reference to each of those certificates by matching the value of the `IssuerSerial` element and checking the digest value in the `ds:DigestValue` element.

References to revocation data are held in the `CompleteRevocationRefs` optional unsigned

property. The two major types of revocation data are CRLs and responses of on-line certificate status servers. The `CompleteRevocationRefs` property can contain sequences of references to CRLs, sequences of references to OCSP responses and other references to any alternative types of revocation data.

The signer's attribute certificates, if any, and corresponding revocation data references are contained in the `AttributeCertificateRefs` and `AttributeRevocationRefs` properties, respectively, which have the same types as the previous ones.

As a minimum, the signer must provide XAdES-BES or XAdES-EPES. If the signer does not provide XAdES-T nor XAdES-C, the verifier should create them as soon as possible. A verifier claiming time-stamped XAdES-C facilities must support:

- Verification of the XAdES-BES form.
- The `SignatureTimeStamp` unsigned property (XAdES-T).
- The `CompleteCertificateRefs` and `CompleteRevocationRefs` unsigned properties.
- X.509 Public Key Certificates as defined in [22].
- Either CRLs [6] or OCSP [7].

2.2.5 Extended Signature Formats

In addition to the four main signature formats, three extended forms are defined regarding signature storage and very long term verification: *Extended Signatures with Time* (XAdES-X), *Extended Long Signatures with Time* (XAdES-X-L) and *Archival Signatures* (XAdES-A). These formats are obtained by adding certain unsigned properties defined in the specification and are not a conformance requirement.

The first extended form, XAdES-X, builds on signatures containing `CompleteCertificate Refs` and `CompleteRevocationRefs` properties, by adding time-stamps over these references to validation data. The time-stamps cover referenced certificates, CRLs and OCSP responses in case of a later compromise of the corresponding issuers' keys. For instance, if a CRL is created after the corresponding issuer's key is compromised, a bad certificate could have been maliciously removed so it could again be used in signature generation. If this CRL is time-stamped it cannot be claimed to exist before the key compromise. Also, the time-stamp prevents the XAdES-C data from tampering, allowing verifiers to be confident on the validation data that should be used.

XAdES-X-L builds up on XAdES-X by adding the `CertificateValues` and `RevocationValues` unsigned properties. When dealing with long term validity, all the data used in the verification must be conveniently archived. Therefore, the `CertificateValues` property contains the full set of certificates that have been used to verify the signature, including the signer's certificate. When both `CompleteCertificateRefs` and `CertificateValues` are present, all the referenced certificates must be present either in the `ds:KeyInfo` element or in the `CertificateValues` element. In addition, the `RevocationValues` property holds the values of all the revocation data used in the verification of the signature. When both `CompleteRevocationRefs` and `RevocationValues` are present, all the referenced revocation data must be present either in the `ds:KeyInfo` element or in the `RevocationValues` element.

Finally, the XAdES-A form builds up on XAdES-X-L by incorporating one or more `ArchiveTimeStamp` unsigned properties, which holds time-stamps over most the signature's structure, namely unsigned properties containing validation data. This form also covers the problem of degradation of the cryptographic algorithms over time if an archive time-stamp is applied on a regular basis. Even if the signature uses some old and weakened algorithms, there is always a time-stamp based on stronger mechanisms that covers the sensitive data.

CHAPTER 3

State of the Art

This project focuses on XML Advanced Electronic Signatures, which by specification build upon the basic XML Digital Signatures. It is therefore appropriate to assay existing Java implementations of the XML-DSIG specification, which could be used by the library being developed. Furthermore, it is relevant to analyze any existing Java solutions or on going projects related to XAdES in order to try to establish a base line for comparison with the current work and to learn some pros and cons that may affect the library's architecture. This chapter presents those analysis: Section 3.1 focuses on the two most relevant implementations of XML-DSIG; in turn, Section 3.2 covers three XAdES projects.

3.1 XML-DSIG Implementations

The Java platform includes a standard API for XML Signatures and a corresponding reference implementation [23]. This API is widespread not only in usage, but also in examples, tutorials and other documentation. Another well-known and stable implementation is provided by the Apache XML Security project [12]. Both have pros and cons, namely when being used beyond their primary purposes.

3.1.1 Java XML Signatures

Since version 1.6 the Java platform includes a high-level implementation-independent API for XML digital signatures services. This API was defined under the Java Community Process as

Java Specification Request 105 [23] and is designed to be independent of specific XML representations, although every implementation must support the default mechanism type: DOM. Furthermore, implementations should rely on JCA service providers, enabling extensibility in components such as cryptographic services, transform algorithms and URI dereferencing.

The core of the Java XML-DSIG API is the XMLSignatureFactory abstract factory [24], which exposes factory methods to create objects that represent all the XML-DSIG elements. In fact, each element defined in the specification has a corresponding interface on the API, based on the XMLStructure interface, which represents a XML structure from any namespace. Each concrete XMLSignatureFactory implementation supports a specific XML representation that identifies the XML processing mechanism used to parse and generate XML structures.

The factory methods on XMLSignatureFactory are passed the content of the elements they create, as illustrated in Listing 3.1 for the `Reference` element.

Listing 3.1: Building a Reference

```
// The factory
XMLSignatureFactory xmlSigFact = XMLSignatureFactory.getInstance("DOM");
// The digest method
DigestMethod dm = xmlSigFact.newDigestMethod(DigestMethod.SHA1, null);
// A transform
Transform transf = xmlSigFact.newTransform(Transform.ENVELOPED, null);
// The reference element
Reference ref = xmlSigFact.newReference("uri", dm,
    Collections.singletonList(transf), null, "refId");
```

This kind of usage is not user friendly, mostly because it is too attached to the XML representation. To create a Reference one previously has to create a DigestMethod and a list of Transforms; this clearly mirrors the XML hierarchy, which the developer has to be reasonably familiar with. Even though the approach enables maximum customization of the signature's structure, the bottom line is that the application developer wants to specify a set of characteristics of the data object reference. The same applies to keying information: the developer wants to specify the public key or certificate without knowing the details of how it gets into the signature. Furthermore, as most signatures will have identical XML structures, that kind of setup code will be repeated whenever a signature is to be generated.

Despite being designed to be mechanism-independent, the API cannot be used regardless of the mechanism in place: at some point one has to bridge between the API and the mecha-

nism being used by the application. For instance, if DOM is being used, the signature has to be appended to an existing node. This node is supplied via a DOM-specific XMLSignContext when invoking the sign method over a XMLSignature, as illustrated in Listing 3.2.

Listing 3.2: DOMSignContext

```
public interface XMLSignature extends XMLStructure {
  void sign(XMLSignContext signContext);
  // ...
}
public class DOMSignContext ... implements XMLSignContext{
  public DOMSignContext(Key signingKey, Node parent) { ... }
}
```

In addition, some elements of XML-DSIG — namely the `Object` element — may have *any* content, which means there has to be a mechanism-specific XMLStructure to incorporate unknown content. For a DOM-based implementation, javax.xml.crypto.dom.DOMStructure is used, which wraps a org.w3c.dom.Node. A quick look at the marshaling method of the DOM-specific XMLObject class [25], shown in Listing 3.3, reveals that when an internal mechanism-specific class is not found on the object's contents, a javax.xml.crypto.dom.DOMStructure is expected.

Listing 3.3: Excerpt of DOMXMLObject marshalling

```
XMLStructure object = (XMLStructure) content.get(i);
if (object instanceof DOMStructure){
  // org.jcp.xml.dsig.internal.dom.DOMStructure
  // Base class for mechanism specific structures
  ((DOMStructure) object).marshal(objElem, dsPrefix, context);
} else {
  // XMLStructure to hold mechanism-specific content
  javax.xml.crypto.dom.DOMStructure domObject =
      (javax.xml.crypto.dom.DOMStructure) object;
  DOMUtils.appendChild(objElem, domObject.getNode());
}
```

Note that this mechanism-specific classes are public and have to be shipped along with the corresponding mechanism implementation. Nevertheless, most of the API can be used independently of the mechanism.

As far as XML signatures are concerned, the API is sufficient. However, when using it as a foundation for further work, some limitations arise concerning the data object references, namely the fact that dereferenced and pre-digest data can only be obtained after a Reference has been generated or validated. This is not necessarily a drawback: its a result of the API being strictly designed for its purpose. Nevertheless, if this data is needed for some additional processing before/during the signature production, one can't access it. Attempts to explicitly dereference same-document References using the default DOM-based implementation of the URIDereferencer interface also failed. This happened because the XMLSignature was not marshalled yet, which only occurs during the sign operation.

Since the API is standard and mechanism-independent, the majority of the classes implementing the default provider are internal. This makes it impossible to directly access some useful base functionality like DOM tree canonicalization and same-document references processing. It would also be helpful if there was a means to reach the DOM nodes corresponding to the signature's elements. This could have been done with a single public interface that would be the base of all the internal XMLStructures. The API could still be mechanism-independent, but it would be for the developer to decide how coupled the application was to a specific mechanism.

3.1.2 Apache XML Security

Apache XML Security [12] is an open project aimed at providing implementations of XML security standards. Currently the project provides Java and C++ libraries that implement the XML Signatures [3] and XML Encryption [26] W3C standards. It has been around since 2001 and has had regular updates since then.

The library is DOM-based and also includes the standard JSR 105 API. Applications can use XML Security as a provider for the standard JSR 105 API or directly use the library's API. In the first case, most of the considerations made in the previous section apply. The library's specific API, however, deserves further analysis. As in the JSR 105 API, there is a type that represents each element in the signature. The main difference is that the types are concrete classes and consequently no factories are needed. Since DOM usage is a principle, all the types expose the corresponding DOM element, which means the developer does not need to explicitly navigate the DOM tree.

The different steps of building a signature's structure are done in a more friendly way than with the Java XML-DSIG API. For instance, one can create an instance of XMLSignature and

then progressively add the different components to that instance, as illustrated in Listing 3.4. The API also simplifies some actions, such as adding References by using the addDocument method. Not only is this more intuitive, because a signature is composed by data object references, but also abstracts the developer from the signature's structure.

Listing 3.4: Apache XML Security API sample

```
XMLSignature xmlSig = new XMLSignature(doc, "",
    XMLSignature.ALGO_ID_SIGNATURE_DSA);
Transforms transforms = new Transforms(doc);
transforms.addTransform(Transforms.TRANSFORM_ENVELOPED_SIGNATURE);
xmlSig.addDocument("", transforms, Constants.ALGO_ID_DIGEST_SHA1);
```

Apart from this shortcuts, the API enables a fine-grained control over the data object references: one can access the referenced data before or after it is transformed anytime during the signature processing. In addition, all the intermediate functionality, such as canonicalization and resource resolving, is available for further use.

The library is also highly configurable and extensible: the different components involved in the signature processing — such as Canonicalizers, ResourceResolvers, MessageDigestalgorithms and Transforms — have known interfaces and are created through factory methods such as Canonicalizer.getInstance(String algorithm). The factory methods resolve the creation of components based on the mappings in a XML configuration file, illustrated in Listing 3.5, where the different implementations of a component are registered.

The main drawbacks of the library are related to documentation — which is short at some points, namely in exception scenarios — and to the exception model, as most of the methods throw general, meaningless exceptions that do not allow the developer to identify their cause. In addition, the support for the algorithms in use is sometimes not validated until they are actually going to be used. For instance, a XMLSignature can be created using an algorithm that is not supported and this will only be noticed when the sign method is invoked. This is not necessarily bad, it just delays the assertion. The problem comes when one can't narrow down the cause of the error due to the poor exception model.

To sum up, the Apache XML Security can provide some useful additional functionality and easy extension/configuration. Nevertheless, it is important to note that one is stepping away from the standard API which means it is not easy to swap to a different XML Signature provider. Also, one should be aware that error handling can be cumbersome.

Listing 3.5: Apache XML Security configuration excerpt

```
<CanonicalizationMethods>
  <CanonicalizationMethod
      URI="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"
      JAVACLASS="org.apache.xml.security.c14n.implementations.Canon20010315"
      />
  ...
</CanonicalizationMethods>
<JCEAlgorithmMappings>
  <Algorithms>
    <Algorithm URI="http://www.w3.org/2000/09/xmldsig#sha1" JCEName="SHA-1"/>
    ...
  </JCEAlgorithmMappings>
 </Algorithms>
```

3.2 XAdES Implementations

During the research for XAdES implementations three open source projects were found. Two of them are mostly individual initiatives while the other is supported by a computer security company. This section provides an analysis of those projects, describing the supported XAdES features, such as formats and signature production and/or verification. Furthermore, there is an overview of each library's API and the resulting usage style.

In addition to those three open implementations, there is a proprietary project undertaken by the Institute for Applied Information Processing and Communications (IAIK) of the Graz University of Technology in Austria. Besides not being open source, it is a commercial product, so it is not considered in this analysis.

3.2.1 jXAdES

The *jXAdES* project [27] was started in 2007 by Miroslav Nachev with the objective of providing a Java implementation of XAdES. The author stated he wished the project could become part of future JDK releases, namely JDK 7. Although many references can be found to the project, its development appears to have been slow and its code base is not very solid yet. The project relies on the JSR 105 API for core XML signatures services and enables generating signatures in the four main XAdES formats. However, verification of XAdES properties is not implemented and no certification path validation is done in the

basic verification process.

Producing a signature involves two main aspects: representing a XAdES form and its properties; and representing an advanced signature, which has a set of qualifying properties (depending on the form) in addition to the core XML-DSIG elements. Each form is represented by an interface that has setters for the different properties allowed in the corresponding form. Instances of types that implement those interfaces are created through a factory method, as illustrated in Listing 3.6. The actual return type corresponds to the XAdES enumerate value supplied to the factory method (the interfaces for the different formats are all based on XAdES_BES).

Listing 3.6: Signature form in jXAdES

```
public interface XAdES_BES{
    public void setSigningTime(Date signingTime);
    public void setSigningCertificate(X509Certificate certificate);
    public void setSignatureProductionPlace(SignatureProductionPlace
        productionPlace);
    public void setDataObjectFormats(List<DataObjectFormat> dataObjectFormats);
    // ...
}
public static XAdES_BES newInstance(XAdES xades, Element baseElement);
```

Having set the desired qualifying properties, one creates an instance of XMLAdvancedSignature passing in the form previously created and applies the signature operation, as exemplified on Listing 3.7. This will generate the qualifying properties' DOM tree — through the internal implementations of XAdES_BES and derivate interfaces — and create the final signature structure using the Java XML Signatures API.

Listing 3.7: Signature creation in jXAdES

```
X509Certificate certificate = ...; PrivateKey privateKey = ...;
XAdES_EPES xades = ...;
XMLAdvancedSignature xmlSignature = XMLAdvancedSignature.newInstance(xades);
List refs = ...; refs.add("#elem1");
xmlSignature.sign(certificate, privateKey, refs, ...);
```

This approach seems to fit the purpose of creating the signature's structure, but it actually has some inconsistencies. For instance, the sign method of XMLAdvancedSignature looks like

it could be invoked multiple times to produce signatures over different data object references. However, the data objects' properties are defined over the XAdES_BES interface, which is only used when creating an instance of XMLAdvancedSignature. Another error prone aspect is that it is developer's responsibility to set the signature policy identifier over the XAdES_EPES interface — which extends XAdES_BES — when creating a XAdES-EPES signature (actually, this also happens with the mandatory properties of the subsequent formats). The policy identifier is mandatory in the EPES form, which means that its presence in the signature could be enforced by the library. In addition, the generation of the property's final data assumes that the identifier is always an URL and attempts to establish a connection; this means that the policy document always has to be an online resource, which is not necessarily true.

When it comes to data object properties, the target `Reference` elements have to be referenced by the property. In *jXAdES* the target `References` are specified by the developer when creating the property, as exemplified in Listing 3.8 for the `CommitmentTypeIndication` property. This means that the developer needs to be able to identify the target `Reference` element and to reference it using its Id.

Listing 3.8: Data object properties in jXAdES

```
public interface CommitmentTypeIndication {
  public void setObjectReference(String objectReference);
  public void setCommitmentTypeId(CommitmentTypeId commitmentTypeId);
  // ...
}
```

However, the developer only has control over the `References`' URIs because this is the only information that can be supplied to the sign method of XMLAdvancedSignature. The `Reference` elements are created internally and — curiously — no Id is used. Even if an Id was used, it would have to be available before the actual signature production because there are **signed** properties under consideration. Even in that case, it would still be tedious for the developer to specify the references' URIs, get the `References` he needs and finally use their Ids in the corresponding data object properties. This low-level approach, leaving much of the work to the developer, is also present on other properties. For instance, objects that represent the `AllDataObjectsTimestamp` property are created with the octet-stream that should be timestamped, which only depends on the data object references. All these aspects make the developer's job harder when they could be systematically solved by the library.

Still regarding the `AllDataObjectsTimestamp`, it is interesting to see the excerpt shown in Listing 3.9, which was taken from a test file of the library.

Listing 3.9: AllDataObjectsTimestamp in jXAdES

```
ArrayList<AllDataObjectsTimeStamp> allDataObjectsTimeStamps = ...;
// TODO: Howto obtain the hash that has to be timestamped
allDataObjectsTimeStamps.add(new
    AllDataObjectsTimeStampImpl("perico".getBytes()));
```

Besides unveiling a flaw of the library, this clearly reflects the JSR 105 limitation on obtaining the `References` octet-streams before producing the signature.

The library is poorly documented and has an unpleasant code-base. One can find hard-coded digest and canonicalization algorithm identifiers (which is also a serious limitation), code that creates properties' XML structure repeated in different parts and some twisted implementation strategies. It clearly is a work in progress in need of serious improvements. It does not solve some of the trickier aspects of the XAdES properties and the API is not convenient, leaving too much responsibility for the developer.

3.2.2 OpenXAdES

Established in February 2001, SK [28] is Estonia's primary and currently the only certification authority, providing certificates to Estonian ID Cards. Its core function is to ensure the reliability of the electronic infrastructure behind the Estonian ID Card project. SK and its partners have developed a digital signature architecture named *DigiDoc*, which includes desktop and web applications to produce and verify electronic signatures with the Estonian ID Card. As part of that initiative, SK started the *OpenXAdES* [29] project which includes a Java library (*jDigiDoc*) for creating and verifying digitally signed files.

The whole architecture — and consequently the library — is based on a XML format defined as part of the initiative: the DIGIDOC-XML [30]. This format is a container for documents and signatures over those documents, as defined by the schema in Listing 3.10.

Each `DataFile` element represents an enveloped or detached file to be signed, while the `Signature` elements are XAdES signatures that cover the files in a `SignedDoc` (the `References` are calculated over those files). As for the signatures, the format defines a profile of the XAdES standard — i.e. a set of properties/content that has to be present — that does not exactly match any XAdES form and places some restrictions on the properties' content.

Listing 3.10: Excerpt of DigiDoc's schema

```
<xsd:element name="SignedDoc" type="SignedDocType"/>
<xsd:complexType name="SignedDocType">
  <xsd:sequence>
    <xsd:element name="DataFile" type="DataFileType" minOccurs="1"
        maxOccurs="unbounded"/>
    <xsd:element ref="ds:Signature" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  ...
</xsd:complexType>
```

For instance, the `CompleteCertificateRefs` and `CompleteRevocationRefs` properties are only used with OCSP responses and OCSP responders' certificates and the signature policy is always implied. Furthermore, the `SigningCertificate` property can only contain the reference for the signer's certificate and no data object properties are supported. One cannot choose the actual XAdES form because the *DigiDoc* profile is always applied.

Being based on DIGIDOC-XML, the library is not only closely coupled with the format but also does not offer support for every detail allowed in the XAdES standard. A signature is represented by the Signature type whose instances are created via an instance of SignedDoc, to which they will be attached. The limitations are also visible in the types that represent the signed and unsigned properties. As exemplified in Listing 3.11, the type for signed properties directly holds the information for the signer's certificate, the only one that will be contained in `SigningCertificate` property.

Since data object properties are not supported, the library does not have to solve some issues on obtaining the data for those properties. It would be interesting to analyze the approach on those issues and the resulting interface for the developer. The same applies for generation of time-stamp properties like `SignatureTimeStamp` and `SigAndRefsTimeStamp`, but the library only provides verification of such properties.

Having in mind the limitations of the format, the library's API has some interesting aspects. As shown in Listing 3.12, the data objects being signed are DataFiles which are added to a SignedDoc. After adding the files, one creates a Signature that will be applied over those files.

With this approach one does not have to handle the details of the signature's structure: the `References` for the `DataFile` elements are internally created as well as the other

Listing 3.11: Signed properties on jDigiDoc

```
public class SignedProperties implements Serializable{
  /** signing time measured by signers own computer */
  private Date m_signingTime;
  /** signers certs digest algorithm */
  private String m_certDigestAlgorithm;
  /** signers cert id */
  private String m_certId;
  /** signers certs digest data */
  private byte[] m_certDigestValue;
  /** signers certs issuer serial number */
  private BigInteger m_certSerial;
  // ...
}
```

Listing 3.12: Creating a signature with jDigiDoc

```
SignedDoc sdoc = new SignedDoc(...);
sdoc.addDataFile(new File("..."), DataFile.CONTENT_EMBEDDED);
sdoc.addDataFile(new File("..."), DataFile.CONTENT_DETATCHED);
X509Certificate cert = ...;
Signature sig = sdoc.prepareSignature(cert,
                  null, // String[] claimedRoles,
                  null); // SignatureProductionPlace
byte[] sidigest = sig.calculateSignedInfoDigest();
byte[] sigval = /* calculate signature value */;
sig.setSignatureValue(sigval);
```

elements in the signature. This includes the qualifying properties elements, namely the `SigningCertificate`, whose structure is generated from the X509 certificate passed into the prepareSignature method. The actual signature value is not produced by the Signature class — which only provides the final digest of `SignedInfo` — so that it can be calculated by any means. It is interesting to note this separation because it enables selecting the signing key/certificate from different sources (smart-cards, key-stores) and independently from the rest of the process. The library includes support for smart-cards, namely through a PKCS#11 driver.

The library includes a configuration file where one can set up the classes that implement

time-stamp verification, OCSP verification and canonicalization. Additional parameters can be configured, including the localization of CA certificates. However, this is done with file paths, which can be restrictive.

The code base is a bit cumbersome, having repeated code, hard coded digest and canonicalization algorithms and string-based XML generation. The library does its job in the context that the authors found appropriate, but it is not sufficient for a generic XAdES support. Nevertheless, the project seems to have good foundations due to its creator and main investor.

3.2.3 WebSign Project

The *WebSign Project* [31], started in 2006, consists of a Java applet that enables the user to sign documents in his browser. The documents are signed with XML-DSIG and the signing keys can be selected from files or hardware tokens such as smart cards. Since 2007 the project includes a XAdES module, which can also be used separately from the applet; it is an alpha version which, as stated by the author, "*is not complete and has a lot of bugs*". The author added that the XAdES module was "*in active development*", but the project has no activity since March 2008. Still, it is possible to do a brief analysis of the existing sources.

The project's strategy is simple: each XAdES property is represented by a Java type, whose instances are created by the developer. Also, there are classes to represent the container elements of the four types of properties (signed/unsigned; signature/data objects) and a container for the whole set of qualifying properties (DOMQualifyingProperties). All the types are DOM-based and most of them have a marshal method that creates the corresponding DOM tree. As illustrated in Listing 3.13, when producing a signature one creates the qualifying properties and assembles the containers according to the XAdES structure.

The DOM tree generation (marshalling) is done during that process and one can get the final qualifying properties node through DOMQualifyingProperties. Actually, the unsigned properties are marshalled when they are added to the corresponding container, because the `UnsignedProperties` element can be added/altered after the signature value calculation. The different types also support being created from a DOM node that holds the respective content. This is what the library does concerning the signature's structure; all the remaining XML-DSIG/XAdES structure — namely the incorporation of the `QualifyingProperties` element and the reference over the signed properties — has to be handled by the developer through the JSR 105 API.

Listing 3.13: Creating qualifying properties with WebSign XAdES module

```
DOMQualifyingProperties qualif = new DOMQualifyingProperties(...);
// Unsigned properties
DOMUnsignedProperties unProps = new DOMUnsignedProperties(...);
DOMUnsignedSignatureProperties usp =
    unProps.getOrCreateUnsignedSignatureProperties();
usp.addSignatureTimeStamp(...);
qualif.setUnsignedProperties(unProps);
// Signed properties
DOMSignatureProductionPlace spp = new DOMSignatureProductionPlace(...);
DOMSignaturePolicyIdentifier policy = new DOMSignaturePolicyIdentifier(...);
DOMSignedSignatureProperties ssp = new DOMSignedSignatureProperties(policy,
    spp, ...);
DOMSignedDataObjectProperties sdo = new DOMSignedDataObjectProperties(...);
DOMSignedProperties sp = new DOMSignedProperties(ssp,sdo);
qualif.setSignedProperties(sp);
```

The project's simple strategy goes on when it comes to the types that represent the different qualifying properties: all the information that is needed in the property has to be created by the developer; the types are very "low level". For instance, the `SigningCertificate` property is created from a list of certificate references, each containing the certificate's digest and issuer/serial, as shown in Listing 3.14. It is also shown that the data object reference in the `DataObjectFormat` property has to be supplied to the corresponding type's constructor.

This approach is present throughout the library's classes, including the ones that represent time-stamp properties, whose data is hard to generate. In fact, with the JSR 105 API it is not even possible due to the References limitation on getting the pre-digested data before the actual signature generation. On the other end, when it comes to verifying a signature, the raw data is all that will be available, since the library only handles the DOM unmarshalling.

To sum up, the *WebSign* XAdES module is just a Java mirror of the XAdES properties' XML structure, handling the DOM marshalling and unmarshalling. It does not handle some XAdES structural rules and leaves all the data gathering to the developer. No considerations can be made about the XAdES formats, signing key selection or components configuration because the library clearly is at a lower level. A lot of extra work would have to be done upon the library in order to produce or verify XAdES signatures.

Listing 3.14: Examples of qualifying properties in WebSign XAdES module

```
public class DOMCertIDList extends DOMStructure{
  public DOMCertIDList(List<CertID> certs) { ... }
  public static class CertID{
    public CertID(DOMDigestAlgAndValue certDigest, DOMX509IssuerSerial
        issuerSerial){ ... }
    // ...
  }
  // ...
}
public class DOMDataObjectFormat extends DOMStructure{
    public DOMDataObjectFormat(String description, DOMObjectIdentifier
        objectIdentifier, String mimeType, URI encoding, URI objectReference)
        { ... }
    // ...
}
```

CHAPTER 4

Architecture

PRODUCING a XAdES signature involves many details, specially if one goes further than the basic form. Besides obeying to the XML structure one needs to gather all the information that makes up the qualifying properties. This information goes from simple strings, such as algorithms identifiers, to calculated digest values, time-stamps and base64-encoded PKI data. In addition, the means of obtaining the needed information frequently depend on the usage scenario. *XAdES4j* aims at making signature production and verification as straightforward as possible, while being flexible enough to accommodate multiple usage scenarios.

This chapter presents the key concepts of the library and how it is defined around them. Section 4.1 presents the concepts that are directly exposed to the developer and the resulting programming model. Section 4.2 discusses the options regarding the core XML-DSIG processing. Finally, Section 4.3 is focused on the processing model for the XAdES qualifying properties.

4.1 Application Programming Model

4.1.1 Profiles

Producing a XAdES signature is a function of the following inputs:

- Signed data objects.

- Data objects properties.
- Signature properties.
- Cryptographic configuration parameters (e.g. digest and canonicalization algorithms).
- Cryptographic keying information (e.g. private keys).

This work classifies these inputs in two groups. The first includes the signature properties and the cryptographic parameters and keying information, which encompass characteristics of the signer/signature. The second group consists of signed data objects and their properties, which define the resources being signed.

Considering the signatory as an entity that produces multiple signatures in a specific context, it is likely that his characteristics are similar on these different signatures, while the signed resources vary. Based on that observation, *XAdES4j* defines the set of invariant characteristics of the signatory and its signatures as a *signature profile*.

A signature profile is used to create a *signer*, i.e. an entity that produces signatures according to the characteristics on that profile, as illustrated in Figure 4.1. On a given context, multiple resources may be signed using the same signer, hence resulting on signatures with similar structure but over different content.

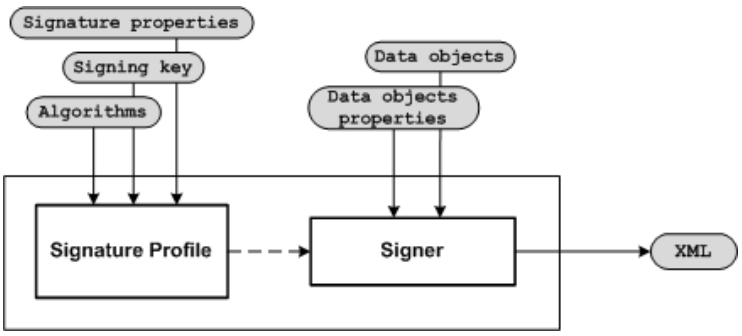

Figure 4.1: Signature production model

A signer is immutable, meaning it will always produce signatures according to the profile's characteristics at the moment of creation. The signature profile, however, may be used to create multiple signers with slightly different features.

A XAdES signature is also defined by its form, i.e., the set of mandatory signature properties. Being a signature's characteristic, the form is also a part of the signature profile. However, different forms require different data: for instance, XAdES-EPES requires information about the signature policy which is not necessary in XAdES-BES. Therefore, the library

includes multiple signature profiles, one for each XAdES form, represented by different Java classes. This a type-safe, explicit way to statically ensure that the developer provides the needed information.

The signature profile concept is also applicable to signature verification. Although there's no need to define signature characteristics, the features of the verification process still have to be defined, namely the cryptographic validation information that should be used. As for signature production, there is a *verification profile* that encompasses the characteristics of the signature verification process. As illustrated in Figure 4.2, it is used to create a *verifier*, i.e., an entity that verifies signatures according to the characteristics of a profile.

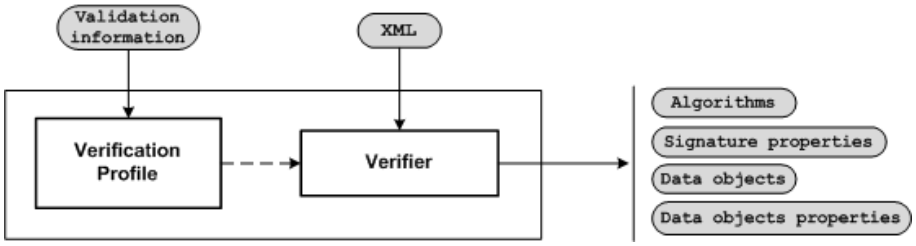

Figure 4.2: Signature verification model

Most of the information that is used as input for signature production is an outcome of signature verification. This highlights the symmetric structure of both processes, which is reflected on the library's internal organization.

4.1.2 Abstraction Level

The library's programming model creates an abstraction over the XAdES specification, the Java platform and other libraries used internally. Different levels of abstraction were considered in the library's design. On one hand, a low-level programming model would expose the XAdES details and the third party APIs. On the other hand, a high-level model would only expose the main concepts, while the structural and processing details are internally ensured.

The JSR 105 API is an example of a low abstraction level, as it mirrors the signature's XML structure. While this enables maximum flexibility, it leaves too much of the signature's structure to be handled by the developer. Adopting a similar approach in *XAdES4j* — with front-end types that reflect the qualifying properties XML structure — would only aggravate the drawbacks in the ease of use, specially because many XAdES properties have several hierarchy levels and relate to XML-DSIG elements. Besides, XAdES has specific structural rules, such as the reference over `SignedProperties`, which would have to be enforced by the

application developer by properly constructing the various elements. Not only is this error prone but also forces the developer to be familiar with the details of XAdES. This approach would be similar to the one in the *WebSign* project [31], which defines a set of low-level building blocks and leaves all the object assembling to be done by the user.

A better approach is to collect the information that should be present in the signature and systematically handle the core signature structure and the XAdES incorporation rules. Note that every XAdES signature has a `SignedInfo` with `Reference`s, including the one over the `SignedProperties` element, and an `Object` where the `QualifyingProperties` element is appended. As an example, Apache XML Security already goes that way with XML-DSIG, since one doesn't need to explicitly create `SignedInfo` or `Reference`s structure. Also, the *jXAdES* project [27] has a similar principle with XAdES: from the developer's perspective, a signature is produced from the set of qualifying properties, a list of references and the keying information; all the structural details are handled internally.

It is important and convenient to internally handle the core structural details, but the same issue arises when it comes to qualifying properties. Most of them have complex XML structures, holding data resulting from previous processing. For instance, the `SigningCertificate` and `CompleteCertificateRefs` properties contain digests of certificates. Furthermore, the time-stamp properties contain time-stamp tokens that result from complex input calculations and subsequent communication with a TSA. It is not helpful to relieve the developer from the core structure and then ask him to handle that kind of details, not only because it is inconvenient and error prone, but also because the XAdES specification clearly defines how to obtain the contents of each property. Thus, *XAdES4j* has the design goal of handling the production of the final property contents whenever possible.

An example of this approach is to represent the `CompleteCertificateRefs` property with a collection of Java's `X509Certificate` instances. During signature production the library calculates the corresponding digests and adds them to the final DOM tree using the appropriate structure. The *jDigiDoc* project [29] has this principles: the OCSP responses and certificates are obtained according to configured parameters and the contents of the `CompleteRevocationRefs` property are internally created. *XAdES4j* extends this behavior to all the eligible properties. The base principle is that the types that represent the qualifying properties and are used by the developer contain *high-level* data, different from the one in the properties' final structure.

The same principle is applied to signature verification: each qualifying property has well

defined verification procedures that need to operate on the detailed property data. These verifications are undertaken by the library — because otherwise the whole purpose would not be fulfilled — and should result on the high-level data, which is easier for the developer to handle.

To sum up, the library is designed to have a front-end that moves away from the signature structure and to undertake all the needed operations to overcome the resulting abstraction.

4.1.3 Service Providers

The XAdES specification defines the information that makes up an advanced signature and how it should be structured and processed. Producing and verifying a signature according to that structure involves a series of tasks for which the outcomes are more relevant than how they're actually achieved, such as:

- Selecting the signing key and certificate.
- Interacting with time-stamping authorities (TSA).
- Selecting signature qualifying properties.
- Selecting algorithms.
- Validating certificates.
- Generating XML.

Some of the outcomes of these tasks have already been identified as inputs for signature production or verification through the corresponding profiles. However, directly depending on the needed information is quite restrictive. For instance, consider a scenario where the keying information is stored on a smart card. If the signing key (actually its in-memory reference) is directly used as an input for the signature profile, the user will have to be prompted for the corresponding PIN before the actual signature operation. A better approach is to have an entity that is responsible for providing the signing key right when it is needed. This means that the signature profile — and consequently the signer — is not depending on the key itself but on a means of obtaining it. Furthermore, the specific actions undertaken to obtain the key are not relevant to the signature producer: it might be obtained from a Java key store or a Windows key store, but that won't change the signing process as a whole. Note that if the key is to be directly provided, one can still create a wrapper that returns it whenever needed.

The approach just described for obtaining the signing key is also used on the other tasks presented above. For instance, instead of directly handling validation information, the verification process relies on an entity that validates certificates. Furthermore, the algorithms involved in the signature are also obtained through one of those entities. A possible realization is asking the user which algorithms should be used; another option is to have a default set of algorithms that is always used in a given context. In *XAdES4j*, the entities that are used to accomplish this tasks are defined as *service providers*.

A service provider has a specific interface and is used during signature production and verification whenever some data/functionality is needed and there are multiple ways of obtaining/implementing it without changing the processes as a whole. The library reflects these degrees of freedom by clearly defining interfaces for the different service providers and allowing their independent configuration. This way, the substitution of a service provider won't interfere with others, making it possible to use different combinations and to accommodate multiple use-cases.

The library is also complete, meaning it includes one or more implementations for each configurable service provider. Some of them are configured by default to make the library's usage straightforward in multiple scenarios. Nevertheless, new providers may be created and plugged into the library, as illustrated in Figure 4.3. The library also benefits from the configuration/extensibility capabilities of the underlying Apache XML Security library [1] and the Java platform itself.

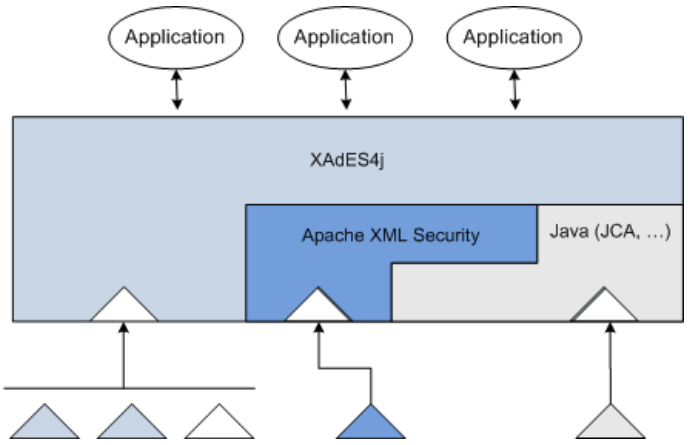

Figure 4.3: Architecture overview

[1] The Apache XML Security library is used to handle the core XML-DISG processing, as described in the following section

The code excerpt on Listing 4.1 illustrates the usage of the library's API, focusing not only the service providers but also the other concepts presented throughout the chapter. The different types on the example are introduced on the following chapter but their names should be sufficient to identify the corresponding concepts.

Listing 4.1: Using the library's API

```
class ExampleAlgorithmsProvider extends DefaultAlgorithmsProvider{
  @Override
  public String getSignatureAlgorithm(String keyAlgorithmName){
      return XMLSignature.ALGO_ID_SIGNATURE_RSA_SHA1;
  }
}
class ExampleSignatureProps implements SignaturePropertiesProvider{
  @Override
  public void provideProperties(SignaturePropertiesCollector spc){
      spc.setSignatureProductionPlace(
        new SignatureProductionPlaceProperty("Lisbon", "Portugal"));
      spc.setSigningTime(new SigningTimeProperty());
  }
}
// ------------------------------------------------------------------
KeyingDataProvider ptccKey = new PKCS11KeyStoreKeyingDataProvider(...);
XadesTSigningProfile p = new XadesTSigningProfile(ptccKey)
  .withAlgorithmsProvider(ExampleAlgorithmsProvider.class)
  .withSignaturePropertiesProvider(ExampleSignatureProps.class);
XadesSigner signer = p.newSigner();
signer.sign(new DataObjectReference("http://..."), ...);
```

4.2 Core XML-DSIG Processing

XAdES builds on XML Signatures by adding qualifying properties as well as the corresponding incorporation and validation rules. Nevertheless, all the XML-DSIG concepts still apply and no changes are made to the core processing rules. Since this project is focused on XAdES, it relies on an existing XML-DSIG implementation to handle those aspects. Regardless of the chosen implementation, the most important issues relate to how will it bias the library's design and how exposed will it be to the outside.

During the first half of the project the XML-DSIG implementation in use was the JSR 105 API. This was a natural choice as the API is bundled with the Java platform and

no additional dependencies are needed. However, it proved itself insufficient due to the limitations on accessing the `References`' data. The input for time-stamp properties that apply to signed data objects is obtained by concatenation of the data resulting from processing some or all the `References` in the signature. Since time-stamp properties over data objects are signed properties, the referenced data has to be obtained prior to the actual signature calculation, which is exactly what the JSR 105 API doesn't support. Furthermore, the types that represent the dereferenced data are not very convenient, namely the NodeSetData interface. This interface is used when a node-set results from `Reference` processing, but there is no other way to create an object representing this type of data. It would be useful because the input to other time-stamp properties — such as `SignatureTimeStamp` — results from canonicalizing XML elements within the signature and canonicalization acts upon a node-set. As a side note, at the time of this writing these aspects are unchanged in the JDK 7 API.

In order to circumvent the limitations of the JSR 105 API, Apache XML Security was adopted. It enables the early dereferencing of `References` and has more convenient representations of the resulting data. This change comes with some drawbacks, namely the worse exception model and the addition of an external dependency to the library. Still, Apache XML Security is the best remaining option. The JSR 105 API was the start line for the project and since it is independent of specific XML mechanisms (e.g. DOM, SAX) this also became a relevant question in the library's design. However, there would be no significant advantages on going that way. One of the main reasons is that a specific implementation of *XAdES4j* would only be feasible if an implementation of XML-DSIG for the same mechanism already existed, since a mechanism-specific XMLStructure is needed to embed the XAdES elements. That's not likely on most scenarios because the default (and required) mechanism is DOM, which has widespread support. Being mechanism-independent would also increase the library's complexity on both the front-end and the internals. Since being mechanism-independent is not a priority and would be unnecessary in most scenarios, the work needs not to focus on that matter and the library is DOM-based. Since Apache XML Security is DOM-based, this decision would probably have been forced later by the adoption of that library.

4.3 XAdES Processing and Extensibility Model

Producing and verifying a signature are seen as symmetric processes: roughly, the outputs of one are inputs for the other (and vice-versa). This approach fits even better if one focuses on the qualifying properties: the production process starts with high-level property representations, generates the appropriate property contents and ends up with the final XML structure. On the other hand, the verification process gets the property information from the XML structure, performs the appropriate verification and outputs the corresponding high-level properties.

Going from the high-level property representations to the final XML structure involves two steps: generating the needed information — the *low-level* data — and creating the DOM tree that contains the generated data. These two tasks are very different, not only due to their actual purpose, but also to their requirements. The first step requires access to the algorithms in use (digest, canonicalization), to message digest engines and even to TSAs. On the contrary, the second step comes down to translating a set of data into the corresponding DOM tree. Therefore, the concept of *property data object* (PDO) was created. A PDO is an intermediate representation of a property and contains its low-level data. These data objects play a major role on the qualifying properties life-cycle, as illustrated in Figure 4.4.

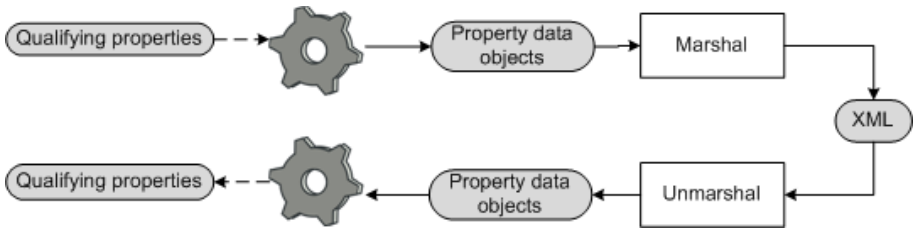

Figure 4.4: Properties life-cycle

PDOs contain all the needed information to fill the properties' DOM tree. However, they don't address any XML-specific issues, such as elements naming and ordering, as those are responsibility of the *marshalling* stage. Marshalling is the process of converting a type that represents a XML structure to the actual XML mechanism. In this case, it is the conversion of a property data object to the corresponding DOM nodes. Unmarshalling is the inverse process.

Although generating low-level properties data and creating the final DOM structure are different tasks, one might argue that they could be done on a single step, without explicitly defining intermediate data objects. However, there are other reasons for their existence. The

first one — which can be seen as a consequence — is that marshalling and unmarshalling become independent from the other steps. This enables exchanging the actual means of creating/parsing the DOM tree without affecting the remaining steps, even using user-defined implementations.

The second main reason concerns the validation of property data. For instance, the `SignatureProductionPlace` property has four optional fields but at least one of them has to be specified. Another example is the `SigningCertificate` property: it must contain at least one certificate reference. These kind of checks have to be done on both signature production and verification, which makes the property data objects the appropriate subject for that task.

XAdES4j includes types to represent XAdES qualifying properties as well as the corresponding PDOs, which are processed according to the model just described. In addition, it includes a set of *property data generators*, which are responsible for creating PDOs for a given collection of high-level properties, and a set of *property verifiers* that perform the opposite transformation while applying the XAdES verification rules.

The properties processing model is extensible, meaning that new qualifying properties can be used. To that end, one should define the high-level property types and the corresponding PDOs. In addition, the appropriate property data generators should be created. If property validation is required, custom property verifiers should also be defined. These items are then plugged into the signature production and verification processes which will employ them as needed.

CHAPTER 5

Implementation

The current chapter describes the library's implementation details, covering the realization of the key concepts and the internals of the different signature services. Section 5.1 introduces the types that represent the qualifying properties and the corresponding data objects (PDO). The following two sections present signature production and verification, namely the profiles and the different stages of those processes. Section 5.4 describes how the library supports extending a signature's form. The details on how the different profiles are processed in order to create the respective signers/verifiers are covered in Section 5.5, while Section 5.6 is focused on the library's exception model. Finally, the last two sections present examples of service providers bundled with the library and some considerations on the library's tests, respectively.

In order to simplify the text, the packages of the different Java types are not referred. Appendix A presents the library's organization and assigns the most relevant types to their respective packages.

5.1 Qualifying Properties

The core of XAdES is a set of qualifying properties that can be combined to make up the different signature formats. These properties can be classified in line with two attributes:

1. Signature coverage: properties may or may not be included in the signature.

2. Property target: properties apply either to the signature or to the data objects being signed.

In *XAdES4j* the qualifying properties are represented by the QualifyingProperty interface, which is the base for the properties class hierarchy shown in Figure 5.1. It provides basic functionality to identify the property's type and name. The leafs in the figure represent the four types of properties (combinations of signature/data objects and signed/unsigned) which are base classes for specific properties, such as DataObjectFormatProperty and SigningTimeProperty. Data object properties, either signed or unsigned, may apply to one, many or all the signed data objects, depending on the property. DataObjectProperty is the base class for those properties and includes logic to validate the number of objects that a property can be applied to.

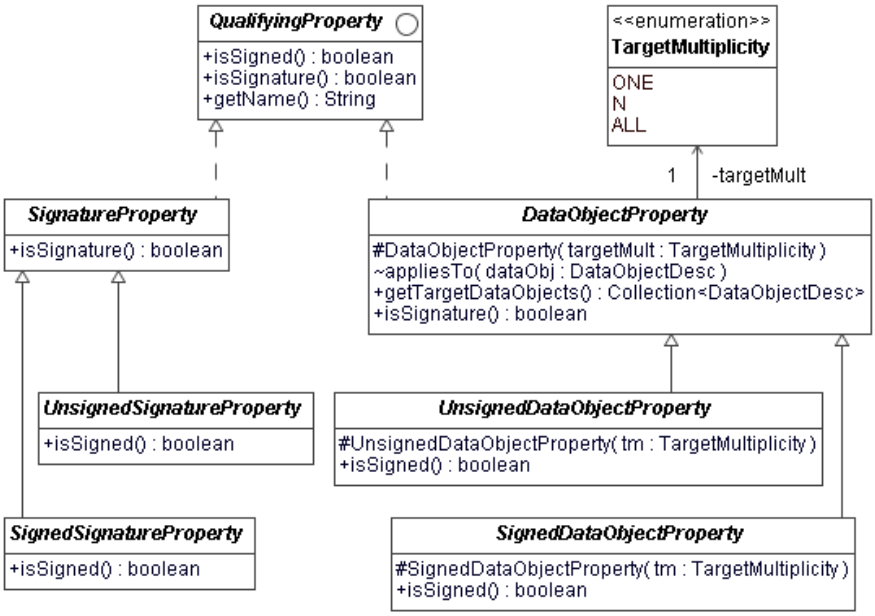

Figure 5.1: Excerpt of the qualifying properties hierarchy

The XAdES schema for signature and data objects unsigned properties is open, meaning that new property elements can be added. To accommodate this possibility and also account for new signed properties that may result from updates to the specification, a class was created to represent each type of custom properties. Each of those classes extends the corresponding leaf class of the hierarchy shown in Figure 5.1. For instance, custom signed signature properties are represented by the OtherSignedSignatureProperty class, which extends SignedSignatureProperty. This approach is preferred to one where the custom properties directly extend the leaf classes in Figure 5.1 because it is simpler to distinguish known and

unknown properties, which in turn eases the control of occurrences of each property.

5.1.1 Property Data Objects

The class hierarchy just described constitutes the high-level representation of the qualifying properties. During signature production those properties are processed in order to obtain the final XML structure. On the other hand, during signature verification they have to be created from XML. As previously discussed, on both cases there's an intermediate representation of each property that contains its low-level data: a property data object (PDO). In the signature production process PDOs are generated from the corresponding high-level properties and their contents are then used to create the final XML structure. On the contrary, in the verification process the data objects are created from the properties XML structure and then processed to obtain the high-level representation.

PDOs are represented by the PropertyDataObject interface, which is a marker interface to group all the data objects. The library includes data object types for all the available properties, plus a GenericDOMData data object that contains a DOM node. It can be used by new or existing properties whenever it is convenient to directly represent the final DOM structure.

One of the purposes of PDOs is to concentrate the structural verification, which is needed on both signature production and verification. Each PDO **must** have a corresponding structure verifier, represented by the PropertyDataObjectStructureVerifier interface shown in Listing 5.1.

Listing 5.1: The PropertyDataObjectStructureVerifier interface

```
public interface PropertyDataObjectStructureVerifier{
  void verifyStructure(PropertyDataObject propData) throws ...;
}
public @interface PropDataStructVerifier{
  Class<? extends PropertyDataObjectStructureVerifier> value();
}
```

Data objects and structure verifiers are closely related since one can't exist without the other. The association between a PDO and the corresponding verifier is established by annotating the PDO class with PropDataStructVerifier.

5.2 Signature Production

As previously discussed, a signature can be seen as having two parts: the first consists of the characteristics of the signer and the signature operation itself; the second, the resources being signed. If the signer is seen as a regular signature producer, he's likely to have a set of characteristics that are used whenever a signature is created, i.e a signature profile. These characteristics are fixed between signatures, while the signed resources vary. Thus, producing a signature is to combine a profile and a set of resources in order to create the final XML structure. This process comes down to three major tasks: gather the needed information (signature and data objects properties, algorithms, keying data) in appropriate order; create the core signature structure using the Apache XML Security API; and create the qualifying properties DOM tree to be appended to the signature. Note that Apache XML Security creates the DOM tree for the core signature structure. However, the XAdES elements are unknown to the Apache API, which means that the last task has to be completely supported by the library.

5.2.1 Signed Data Objects and Their Properties

First of all, in order to create a signature one needs to identify the resources being signed. In *XAdES4j* a signed data object is represented by the DataObjectDesc class. It allows the characterization of a data object in terms of transformations — to be applied as part of reference generation — and qualifying properties, as illustrated in Figure 5.2. In XMLDSIG, data objects are identified via same-document or external URI references that fit for enveloped and detached signatures. However, for enveloping signatures it's also necessary to specify the content to be included inside the signature. Thus, the signed data objects are actually specified by the DataObjectReference and EnvelopedXmlObject classes: the first one is used to specify URI references, while the second is used for enveloped XML content. If the content to be enveloped is not XML, it may be encoded as a string and stored in a text node.

When an enveloping signature is being produced the appropriate `Object` elements have to be created to hold the content being signed. Also, there has to be a `Reference` that covers each of those elements so that they are actually included in the signature calculation, which means that the `Object`s need Ids. On the other hand, if any properties are applied to the signed data objects, the corresponding `Reference`s have to be referenced from the properties' elements. The developer doesn't need to handle all those details. Instead, he focuses on the concepts (URI reference, enveloped content, transforms, properties) and describes the

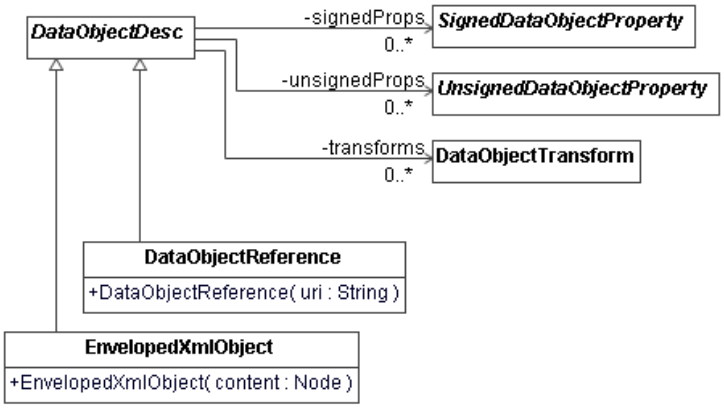

Figure 5.2: Data objects descriptions

resources being signed. For instance, the data object properties are not part of the `Reference` elements in the signature but conceptually they are related to the signed data objects, hence being defined together. During the signature production process the library will handle the data object descriptions and create all the needed elements.

All the actions over the DataObjectDesc class are done via a fluent interface with one method per property type, in opposition to having a single method that accepts an instance of DataObjectProperty, as illustrated in Listing 5.2. If no qualifying properties are specified, they can be added during the actual signature production through a service provider.

Listing 5.2: Interface of DataObjectDesc

```
public abstract class DataObjectDesc{
  public DataObjectDesc withTransform(DataObjectTransform t){ ... }
  public DataObjectDesc withDataObjectFormat(DataObjectFormatProperty f){ ... }
  public DataObjectDesc
      withOtherDataObjectProperty(OtherUnsignedDataObjectProperty op){ ... }
  // ...
}
// ------------------------------------------------------------------------
DataObjectDesc obj = new DataObjectReference("#root").
    .withTransform(new DataObjectTransform(Transform.XPATH, ...))
    .withDataObjectFormat("text/xml")
    .withDataObjectTimeStamp();
```

This results not only in more expressive code but also on an easier control of the number of occurrences of each property. For some properties, a single data object may be the target of multiple property instances. On the other hand, there are properties which can have only

one instance applied to a single data object. Moreover, some data object properties apply to a single data object, while in others cases the same property instance may be applied to multiple objects. Thus, two checkpoints are needed: the first on the DataObjectDesc class, to ensure that the number of instances of each property is correct; the second on the DataObjectProperty class, to ensure that a property is not applied to more data objects than it should. As illustrated in Figure 5.3, these two classes are closely related to ensure that the properties are correctly applied.

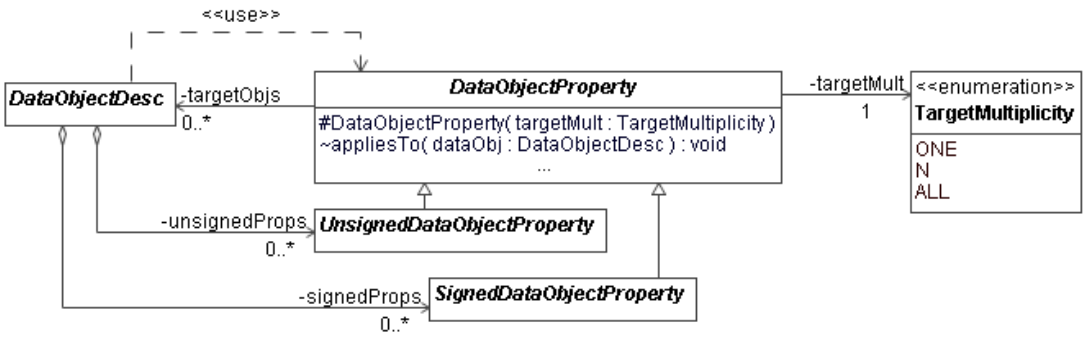

Figure 5.3: DataObjectDesc and DataObjectProperty

The properties are applied to data objects and not otherwise, so the process starts in one of the with... methods of DataObjectDesc. Internally, the first checkpoint is done through the PropertiesSet helper class, shown in Listing 5.3.

Listing 5.3: PropertiesSet class

```
class PropertiesSet<T> {
  private final Map<Class, Set<T>> properties;
  void put(T prop) { ... }
  void add(T prop) { ... }
  void remove(T prop) { ... }
  // ...
}
```

The properties are stored by type and the control of occurrences is made through the add and put methods: the add method allows passing properties of types for which there are already multiple instances; on the other hand, the put method will throw an exception if an attempt is made to add a property of a type for which there is already an instance in the bag. Both methods will also throw an exception if an attempt is made to add a repeated instance. The PropertiesSet class is used throughout the library whenever this behavior is needed.

DataObjectDesc uses two instances of PropertiesSet, for signed and unsigned properties, respectively. The methods on DataObjectDesc will invoke add or put, depending on the property being added, to ensure the correct number of property instances.

Adding a property to DataObjectDesc will end up invoking the appliesTo method over the property itself, passing the current DataObjectDesc instance. This is the second checkpoint: an exception is thrown if the property cannot be applied to more data objects or is already applied to the given data object, as illustrated on Listing 5.4.

Listing 5.4: DataObjectProperty.appliesTo method

```
void appliesTo(DataObjectDesc dataObj){
  if (this.targetDataObjs.size() == this.targetMultiplicity.multiplicity)
    throw new PropertyTargetException("...");
  if (!this.targetDataObjs.add(dataObj))
    throw new PropertyTargetException("...");
}
```

The checks are made using the property's TargetMultiplicity — which indicates the maximum number of data objects that a property can be applied to — and a Set that holds the target data objects descriptions. The DataObjectDesc instances are also stored because the signature generation process is property-centric and it is convenient to easily know the targets of a property.

The DataObjectDesc class allows the characterization of a single data object, particularly its qualifying properties. However, XAdES includes properties that apply to all the data objects, namely `AllDataObjectsTimeStamp`. Therefore, the final set of data objects is represented by the SignedDataObjects class which also enables applying global data object properties. Listing 5.5 exemplifies the characterization of a set of data objects using the classes previously described. The resulting instance of SignedDataObjects will be the actual input for signature production.

5.2.2 Signature Profiles and Signature Producers

A signature profile has been defined as a set of invariant characteristics of the signatory and the signature, used to create a *signer*, i.e., an entity that creates signatures with those characteristics. In *XAdES4j* a *signer* is represented by the XadesSigner interface, shown in Listing 5.6.

A XadesSigner produces a signature over a set of data objects and appends it to a DOM

Listing 5.5: Defining signed data objects and their properties

```
AllDataObjsCommitmentTypeProperty globalCommitment =
    AllDataObjsCommitmentTypeProperty.proofOfApproval();
CommitmentTypeProperty commitment = CommitmentTypeProperty.proofOfCreation();
DataObjectDesc obj1 = new
    DataObjectReference("http://www.ietf.org/rfc/rfc3161.txt")
  .withTransform(new DataObjectTransform(Transforms.Enveloped))
  .withDataObjectFormat(new DataObjectFormatProperty("text/plain"))
  .withCommitmentType(commitment);
DataObjectDesc obj2 = new EnvelopedXmlObject(envelopedElem, "text/xml")
  .withCommitmentType(commitment);
SignedDataObjects dataObjs = new SignedDataObjects()
  .withSignedDataObject(obj1)
  .withSignedDataObject(obj2)
  .withCommitmentType(globalCommitment);
```

Listing 5.6: The XadesSigner interface

```
public interface XadesSigner{
  public XadesSignatureResult sign(SignedDataObjects objs, Node parent);
}
```

Node, returning some information about the generated signature, namely the final set of qualifying properties. The SignedDataObjects instance provides one part of the information needed to create the signature. The other part — signer/signature characteristics — is intrinsic to the XadesSigner in use and depends on the configuration, i.e the profile, that was used to create it. A signature profile is based on the XadesSigningProfile class, shown in Listing 5.7, which, ultimately, is a factory of XadesSigner.

Listing 5.7: Excerpt of the XadesSigningProfile class

```
public abstract class XadesSigningProfile{
  public final XadesSigner newSigner(){...}
  // ...
}
```

Configuring a profile consists on identifying the service providers that should be used by the resulting XadesSigner during signature production whenever it needs a "service". Some

of the providers have the purpose of obtaining characteristics of the signer and the signature, while others actually implement parts of the process. Examples of providers used in signature production and their corresponding interfaces are:

- Signing key/certificate selection — KeyingDataProvider.
- Algorithms selection (signature, digest and canonicalization) — AlgorithmsProvider.
- Indication of optional signature properties — SignaturePropertiesProvider.
- Obtaining time-stamp tokens — TimeStampTokenProvider.

These and other service providers are defined through the XadesSigningProfile as illustrated in Listing 5.8. Together they define *how* the signature will be generated, as the resulting XadesSigners will rely on them.

Listing 5.8: Configuring service providers on XadesSigningProfile

```
public abstract class XadesSigningProfile{
  protected XadesSigningProfile(KeyingDataProvider kp){...}
  public XadesSigningProfile withAlgorithmsProvider(AlgorithmsProvider ap){...}
  public XadesSigningProfile
      withSignaturePropertiesProvider(SignaturePropertiesProvider spp){...}
  // ...
}
```

One of the goals of XadesSigningProfile is that only the service providers that need to be customized are specified in order to ease the developer's work in some straightforward scenarios. All the others will be resolved to default implementations included in the library. For instance, there's a default implementation of AlgorithmsProvider that directly returns the URIs of commonly used algorithms. Another possible implementation is one that prompts the algorithms to the user. Nevertheless, the KeyingDataProvider is mandatory because it's up to the signer to select the signing key and certificate. This interface, shown in Listing 5.9, is used to get the signing certificate or certificate chain during the signature generation process. Furthermore, it is used to get the private key for the given certificate to actually perform the signature operation.

The reason for having two separate methods for getting the certificate and the private key is that this way the key is only used for as brief as possible. In addition, the getSigningKey is passed the certificate because the KeyingDataProvider might manage multiple keys.

Listing 5.9: The KeyingDataProvider interface

```
public interface KeyingDataProvider{
    List<X509Certificate> getSigningCertificateChain();
    PrivateKey getSigningKey(X509Certificate signingCert);
}
```

The SignaturePropertiesProvider service also deserves further notice. It is used to get optional signed and unsigned signature properties (these properties apply to all signature formats). Instead of having multiple getters that would return null whenever the corresponding properties should not be used, the interface has a single method that is passed a SignaturePropertiesCollector, as illustrated in Listing 5.10. When the SignaturePropertiesProvider is invoked it should in turn invoke the appropriate methods on SignaturePropertiesCollector to register the desired signature properties.

Listing 5.10: Collecting optional signature properties

```
public interface SignaturePropertiesProvider{
  public void provideProperties(SignaturePropertiesCollector signedPropsCol);
}
public interface SignaturePropertiesCollector{
  public void setSigningTime(SigningTimeProperty sigTime);
  public void addCounterSignature(CounterSignatureProperty counterSig);
  // ...
}
```

The signature profile is also the appropriate place to set the signature form. Since each form has a specific set of mandatory properties, the inclusion of such properties is enforced by the XadesSigner. However, configuring the signature form is not that straight because the mandatory properties on each form demand specific data. For instance, in XAdES-C all the validation data has to be available, but that shouldn't be an issue when configuring a profile for XAdES-BES signatures. Therefore, there is a profile type corresponding to each XAdES form, as illustrated in Figure 5.4. This is also why the XadesSigningProfile class is abstract: it provides configuration for the common service providers, but not for the form-specific ones.

Each profile adds support for configuring service providers that supply the data needed in the corresponding XAdES form. For instance, the XadesEpesSigningProfile needs a SignaturePolicyInfoProvider in order to obtain the signature policy information. When a XadesSigner

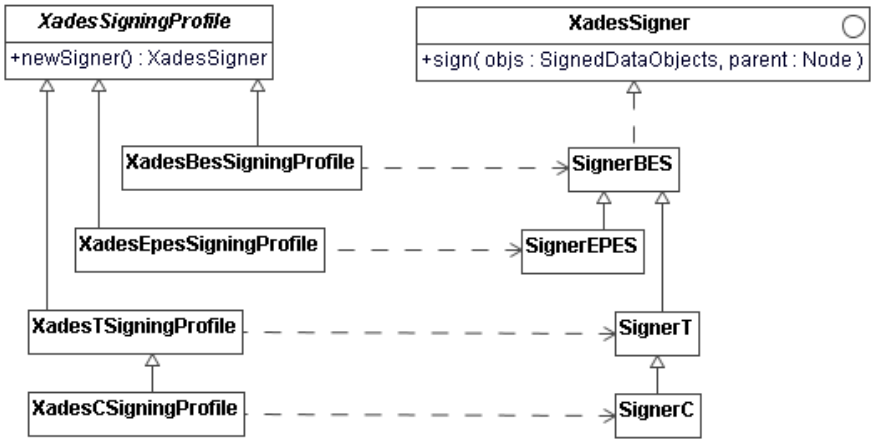

Figure 5.4: The different signature profiles

is requested through the newSigner factory method, each profile will create an instance of the corresponding XadesSigner implementation. In this process, the needed providers are supplied to the signer instance. The details on how this is done are not relevant at this point, but it is worth noting that some of the providers may be used by internal classes besides the XadesSigner implementations.

Despite the existence of multiple signer classes, the core of the signature production process is done by the SignerBes class, as the XAdES-BES form is the base for all the others. The subclasses are only responsible for supplying the form-specific properties. The existing signers support the production of signatures up to the XAdES-C form. Extended forms are also supported but only as part of signature form enrichment, namely right after a verification.

Finally, two important principles apply to the design of the SignerBes class (and consequently its subclasses):

1. **Inversion of Control** — the data and services needed throughout the signing process are supplied by the different service providers. However, the signer is not responsible for obtaining these providers; instead, they are supplied on its construction. The previous description already pointed at this principle.

2. Being **stateless** — each signature operation is independent from the others. The sign method can be invoked multiple times over the same instance and the needed data will always be freshly collected.

5.2.3 Signature Production Core

Having the signed data objects descriptions and the service providers from the signature profile one can get all the data needed to create a signature. The SignerBES class is the center of the signature production process: it creates the whole signature structure, invoking the available services as needed. The final goal is to obtain a DOM tree containing the signature's XML elements. The core XML-DSIG elements are handled by Apache XML Security: one creates instances of types that represent the signature elements and the DOM tree is created in the background. On the other hand, the DOM tree for the XAdES elements is created by the library, as illustrated in Figure 5.5.

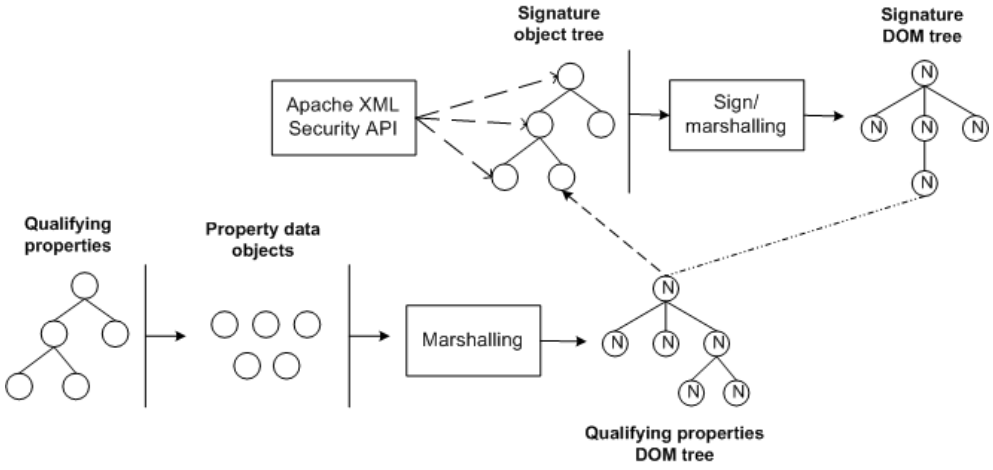

Figure 5.5: Incorporating the qualifying properties into the signature

The signature's structure is not complete until the qualifying properties are appended, which means that the corresponding DOM tree has to be created before the actual signature operation. As previously discussed, getting from the QualifyingProperty instances to the final XML structure will include creating the intermediate PDOs.

The signing operation is focused on three main aspects: the signed data objects and `References`; the keying information and `KeyInfo`; and the qualifying properties. The process is accomplished in several steps, as illustrated in Figure 5.6. It starts with the generation of a unique identifier for the signature, which is also used as a prefix for identifiers on some child elements, namely the `References`. Having a unique identifier is important in case multiple signatures are appended to the same XML document.

The next step is to create an instance of XMLSignature. Unlike the JSR 105 API, in Apache XML Security the signature elements are added to an existing instance of XMLSignature whose structure is progressively built until the sign method is invoked. At this point, the signature

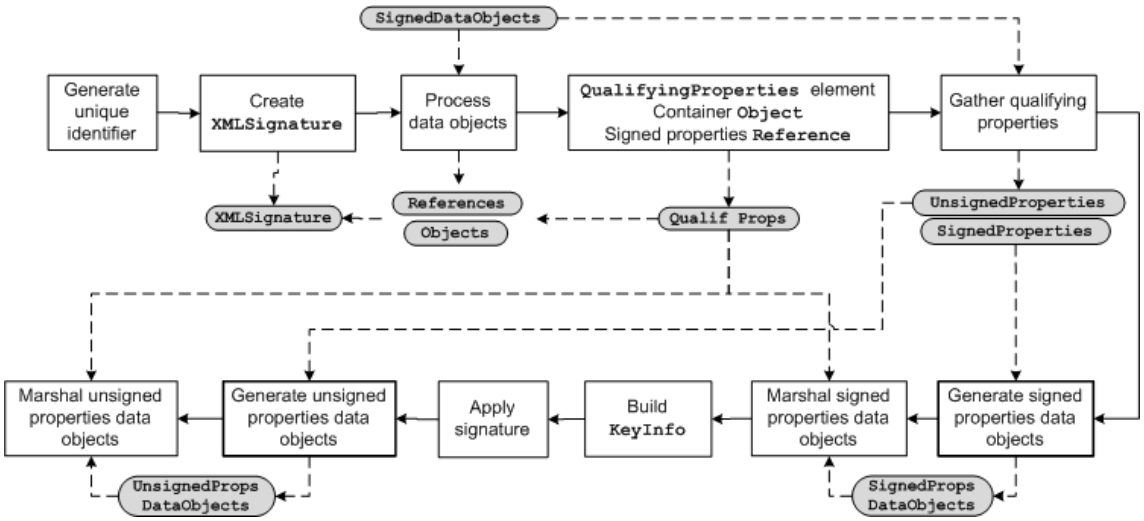

Figure 5.6: Signature production steps

and canonicalization algorithms are acquired from the AlgorithmsProvider in use.

After creating the XMLSignature the focus goes into the descriptions of the resources being signed: the DataObjectDesc instances supplied to the XadesSigner have to be processed in order to create the corresponding `References` and `Objects`. The actions taken upon a data object description depend on its type (URI reference or enveloped content):

- DataObjectReference — a `Reference` is added to the signature, having its `URI` attribute set to the URI reference obtained from the data object description.
- EnvelopedXmlObject — an `Object` element containing the DOM node in the data object description is added to the signature. In order to include the `Object` in the signature calculation a `Reference` is also added to the signature. Its `URI` attribute contains a same-document URI reference using the `Object`'s Id.

During this stage, if any transforms are specified in the data object descriptions they are also added to the corresponding `References`. In addition, the `Reference` and `Object` elements are created with sequential identifiers, prefixed by the signature's ID (they are also unique). These identifiers are needed because, one way or another, each signed data object property will have URI references to the target `Reference` elements. This also means that the identifiers have to be available forth in the signing process, namely when the properties' low-level structure is generated. Since the properties' targets are specified by instances of DataObjecDesc (recall the association between this class and the DataObjectProperty class), some kind of mapping is needed from the data object descriptions to Reference elements.

This mapping has to be built in the current stage because it is where the relevant elements are being created. The adopted strategy is mapping each instance of DataObjectDesc to the corresponding Reference instance (from the Apache API), using an identity Map that will be available when the PDOs are created.

The following step is to create the elements that incorporate the qualifying properties into the core XML signature. This includes creating the `QualifyingProperties` element and appending it to a new `Object`, which is then added to the XMLSignature (the signatures are always created using direct property incorporation). The `QualifyingProperties` element will be used as the parent node when generating the qualifying properties' XML structure. At this stage, the `Reference` over the `SignedProperties` element is also added to the signature. It contains a same-document URI reference using the identifier that will be set on that element.

Having the `QualifyingProperties` element, it's time to create the XML structure of the properties themselves. At this point, the properties are separated in signature properties — through SignaturePropertiesProvider — and data object properties — through DataObjectDesc and SignedDataObjects. However, due to the XAdES XML structure, they have to be gathered in signed and unsigned properties. The properties are grouped using the QualifyingProperties class, illustrated in Figure 5.7.

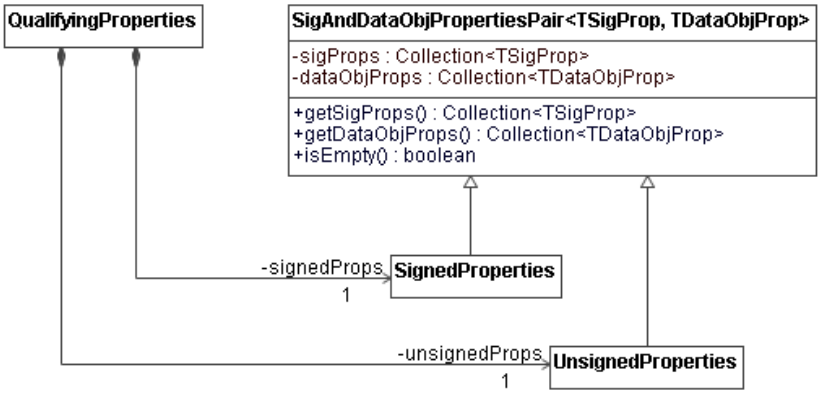

Figure 5.7: Qualifying properties

The separation of data object properties is straightforward, because the DataObjectDesc and SignedDataObjects classes already separate them. If no properties were specified directly over a DataObjectDesc, the DataObjectPropertiesProvider is invoked to add any properties to that object. This is another service provider that can be configured through the signature profile. By default no properties are added in this situation.

The next step is to create the PDOs for signed properties (and only those) and marshal them to obtain the final DOM structure. The separation between signed and unsigned properties comes from their very nature: signed properties need to be marshalled before the signature so that they are covered by a reference; consequently, they can only use information that is available at that point, such as `Reference` elements. On the other hand, unsigned properties do not need to be marshalled before the signature and most of the times they can't be, because they use information that results from the signature generation, like the SignatureValue element. Generating PDOs and their subsequent marshalling are the most important steps towards the final XAdES signature. As such, they are further discussed in upcoming sections.

The XAdES specification doesn't include any rule specifically concerning the structure of the KeyInfo element. Nevertheless, a set of data related to the signing certificate is included in order to ease the signature verification, namely: the certificate itself, the subject name, the issuer name and certificate serial number and the certificate's public key.

At this point, all the information needed to calculate the signature value is available: the signing key is acquired from the KeyingDataProvider and the sign method on XMLSignature is invoked. Finally, the signature's DOM tree is completed by generating and marshalling the unsigned properties data objects.

5.2.4 Generating Property Data Objects

The actions undertaken as part of PDO generation go from direct data copy to digest calculation and time-stamp token attainment. For instance, the high-level representation of the `SigningCertificate` property includes a set of X509Certificate instances. However, the property's XML structure is composed by certificate references (digest and issuer/serial), which are included in the data object for this property, as illustrated in Listing 5.11.

There are three main aspects to account for when generating PDOs:

1. New qualifying properties (not included in the library) may be used and the corresponding data objects have to be created.

2. Creating a PDO may require access to data of the ongoing signing process, namely to `References`, keying information and algorithms in use.

3. The process may require services, namely ones that are already represented by providers in the library.

Listing 5.11: SigningCertificate data object

```
public class CertRef{
  public String digestAlgUri;
  public byte[] digestValue;
  public String issuerDN;
  public BigInteger serialNumber;
}
public class BaseCertRefsData implements PropertyDataObject{
  private final Collection<CertRef> certRefs;
  // ...
}
public final class SigningCertificateData extends BaseCertRefsData{}
```

The generation of a PDO is based on the PropertyDataObjectGenerator generic interface, shown in Listing 5.12.

Listing 5.12: The PropertyDataObjectGenerator interface

```
public interface PropertyDataObjectGenerator<T extends QualifyingProperty>{
  PropertyDataObject generatePropertyData(T prop,
      PropertiesDataGenerationContext ctx);
}
```

A PropertyDataObjectGenerator handles the creation of data objects for properties of a specific type and will be invoked multiple times during the signing process if there are multiple instances of a property. If data of the ongoing signing process is needed, it can be accessed through the PropertiesDataGenerationContext class. The library includes default generators for all the supported properties, but they may be overridden through the signature profile. Furthermore, generators for new properties are also added through the signature profile. These generators may return instances of existing data object types, namely the GenericDOMData or use custom types.

The starting points for this process are the instances of SignedProperties and UnsignedProperties previously created. Two steps have to be taken: the first is to iterate the properties' collections and generate each PDO; the second is to verify the structure of the generated data objects, to ensure that the data supplied in the high-level property representations is valid. For instance, the SignatureProductionPlace property has four optional fields, but at least one of them has to be present. That kind of verifications has to be done at this point because the

generation of data objects may be out of the library's control. The life cycle of the properties can then be represented by the diagram in Figure 5.8.

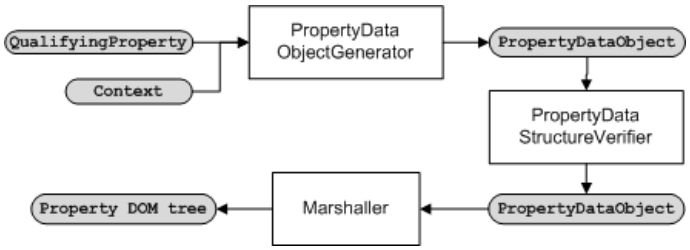

Figure 5.8: Properties life cycle

Even though it is internal functionality, the generation of PDOs is also implemented by a service provider that is supplied to the SignerBES instance. All its implementation does is iterate the properties and obtain a PropertyDataObjectGenerator for each of them. This is done through the PropertyDataGeneratorsMapper interface, shown in Listing 5.13, which provides the data object generator suitable for a given property.

Listing 5.13: The PropertyDataGeneratorsMapper interface

```
interface PropertyDataGeneratorsMapper{
  <T extends QualifyingProperty> PropertyDataObjectGenerator<T> getGenerator(T
      p);
}
```

The purpose of this interface is to make the generation of data objects independent of how the individual generators are actually obtained. This is still internal behavior but it makes the whole process more modular. A possible implementation of the mapper is one that has static associations between the property types and the data object generators. Another option, which is the one in use, is to get the generators from the signature profile, encompassing all the custom configurations.

If a PDO generator needs some functionality that is already defined as a service provider in the library, inversion of control is also applied on its design, which means that the needed providers are constructor dependencies. For instance, a generator of data objects for a time-stamp property will need a TimeStampTokenProvider. All this inversion of control should already point at an important implementation detail which is worth unveiling now: the signature profile uses dependency injection to assemble the different service providers. Thus, the PropertyDataGeneratorsMapper is actually asking the dependency container for instances

of PropertyDataObjectGenerator<T>. Section 5.5 provides details on the usage of dependency injection to resolve all those requests considering the profile configurations.

The PDOs resulting from this process are then returned to the SignerBES instance, which will proceed to their marshalling.

5.2.5 Marshalling Property Data Objects

The last step of qualifying properties processing is marshalling the PDOs into the final `QualifyingProperties` node. This is done separately for signed and unsigned properties at the appropriate time during the signing process. The marshalling process is also represented by service providers that can be configured in the signature profile. Their interface is PropertiesMarshaller, shown in Listing 5.14, which is used for both signed and unsigned properties marshalers.

Listing 5.14: The PropertiesMarshaller interface

```
public interface PropertiesMarshaller{
  public void marshal(SigAndDataObjsPropertiesData props, String propsId, Node
      qualifyingPropsNode);
}
```

The marshalers are passed a pair of collections of PDOs — resulting from signature and signed objects properties — and the `QualifyingProperties` node. Since the PDOs already contain all the needed data, the marshaler's job is to create the appropriate DOM elements and set their content. This includes adjusting some data types, namely by doing base-64 encoding. Note that aspects such as elements hierarchy and order are handled exclusively at this stage.

As for other service providers, there are default implementations of property marshalers, which rely on JAXB [32] to actually perform the marshalling. As illustrated in Figure 5.9, their main task is to convert the PDOs into the appropriate JAXB objects, whose types were generated from the XAdES schema. After the object tree is created, it is passed to the JAXB marshaler to be marshalled into the qualifying properties node.

Although it is an additional indirection, JAXB handles all the type conversions and elements creation with the correct naming and ordering. Nevertheless, this is just one possible implementation that can be exchanged through the signature profile.

The core of the default marshalling process is implemented by the BaseJAXBMarshaller

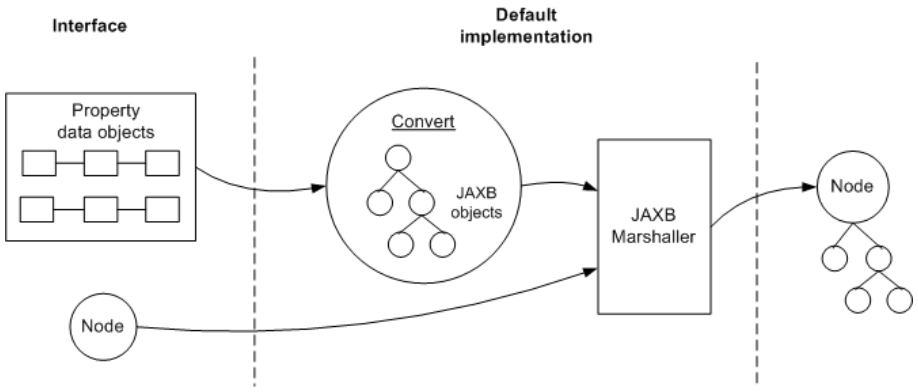

Figure 5.9: Overview of property data objects marshalling

class, shown in Listing 5.15. The TXml type parameter defines the JAXB class that corresponds to the given type of properties (for instance, `SignedProperties`). Note that this element will contain a child element for signature properties and another for data object properties. There are two subclasses of BaseJAXBMarshaller, one for each type of properties, that complete the base class behavior by specifying the TXml parameter and implementing a few abstract methods that depend on the type of properties being handled.

Listing 5.15: The BaseJAXBMarshaller class

```
abstract class BaseJAXBMarshaller<TXml>{
  private final Map<Class, QualifyingPropertyDataToXmlConverter<TXml>>
      converters;
  protected void doMarshal(SigAndDataObjsPropertiesData properties, Node
      qualifyingPropsNode, TXml xmlProps){...}
  // ...
}
interface QualifyingPropertyDataToXmlConverter<TXml>{
    void convertIntoObjectTree(PropertyDataObject propData, TXml xmlProps);
}
```

For each known PDO type there is a converter (a class that implements QualifyingPropertyDataToXmlConverter) responsible for creating the corresponding JAXB objects and by inserting them into the JAXB object tree through the TXml instance. The marshaller iterates the PDOs and invokes the appropriate converter for each object's type.

The default marshalers support all the PDOs supplied with the library, including GenericDOMData, whose content is directly added to the DOM tree after the JAXB marshalling. If other PDOs need to be included a new implementation of the marshalers should be created

and configured in the signature profile.

5.3 Signature Verification

Verifying a XAdES signature is, first, to apply the core XML-DSIG verification rules, which will include verifying the `Reference` over the `SignedProperties` element. Then, the set of qualifying properties has to be inspected in order to identify the signature form and each property has to be verified according to its specific rules. Prior to those verifications, the signature has to be unmarshalled from its DOM tree. As in signature production, Apache XML Security handles unmarshalling the core signature structure. However, qualifying properties elements have to be handled by the library.

5.3.1 Verification Profile and Signature Verifier

Following the principles of signature production, a verification profile is a configuration for a *verifier of signatures*. In *XAdES4j* this entity is represented by the XadesVerifier interface, shown in Listing 5.16.

Listing 5.16: The XadesVerifier interface

```
public interface XadesVerifier{
  public XAdESVerificationResult verify(Element sigElem);
}
```

A XadesVerifier verifies a signature contained in the supplied DOM element and returns a set of information about that signature, namely its qualifying properties and signed data objects. Most of the data needed to verify the signature should be contained in the signature itself but some addition information is needed, namely the trust-anchors that may be considered when validating the signature's certificate. This kind of information is intrinsic to the XadesVerifier in use and depends on the configuration, i.e the profile, that was used to create it. A verification profile is based on the XadesVerificationProfile class, shown in Listing 5.17. Its ultimate purpose is to create instances of XadesVerifier through the newVerifier method.

Configuring a profile consists on identifying the service providers that should be used by the resulting XadesVerifiers during signature verification whenever they need a service. Some of the providers have the purpose of obtaining data needed to verify the signature, but most of them actually implement parts of the process. Examples of service providers and

Listing 5.17: Excerpt of the XadesVerificationProfile class

```
public final class XadesVerificationProfile{
  public final XadesVerifier newVerifier(){...}
  // ...
}
```

corresponding interfaces are:

- Validation of certificates (signature, TSA) — CertificateValidationProvider.
- Obtaining signature policy documents — SignaturePolicyDocumentProvider.
- Verification of time-stamp tokens — TimeStampVerificationProvider.

These and other providers are defined in the XadesVerificationProfile, similarly to the signature profiles. As in signature production, the library includes implementations for the service providers used in signature verification. Most of them are used by default, except for the CertificateValidationProvider implementation because it's up to the verifier to select the actual validation data/mechanism. This provider, illustrated in Listing 5.18, is used whenever a certificate needs to be validated, namely the certificate whose public key will be used to verify the signature.

Listing 5.18: The CertificateValidationProvider interface

```
public interface CertificateValidationProvider{
  ValidationData validate(X509CertSelector certSelector,
      Collection<X509Certificate> otherCerts);
}
```

Despite being identified in the `SigningCertificate` property, the signing certificate itself may not be present in the signature. To account for this situation, the CertificateValidationProvider is passed a X509CertSelector whose selection criteria may be based on information other than the certificate itself, such as the certificate's issuer name and serial number. These items are usually present in the `KeyInfo` element. Nevertheless, if any certificates are present on that element, they are also supplied to the provider. As a result of the validation, the provider returns the certification path and the revocation data used on its validation (only CRLs are supported). Note that it is the provider's responsibility to obtain trust-anchors and/or other intermediate certificates.

When a XadesVerifier is requested through the newVerifier method, the profile creates an instance of the internal XadesVerifierImpl class, which actually implements the verification process, as illustrated in Figure 5.10. In this process, the needed service providers are supplied to the verifier instance.

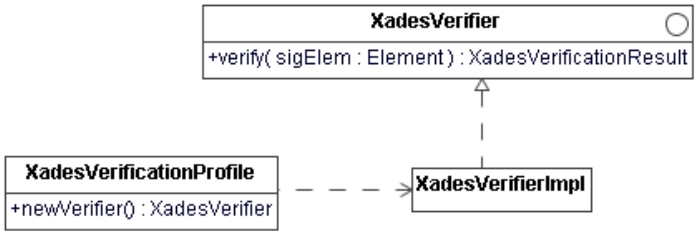

Figure 5.10: Verification profile and signature verifier

The same principles that apply to the design of the signature producing classes (inversion of control and being stateless) apply to XadesVerifierImpl.

5.3.2 Verification Core

The XadesVerifierImpl class is the skeleton of signature verification: it combines the various verification steps and invokes the different services as needed. The verification process is focused in three main aspects: validating the signer's certificate; applying the core XMLDISG verification; and verifying the qualifying properties.

The verification process starts by creating a XMLSignature instance based on the DOM element supplied to the verifier, as illustrated in Figure 5.11. This will unmarshal the signature and its whole structure will become available through the Apache API.

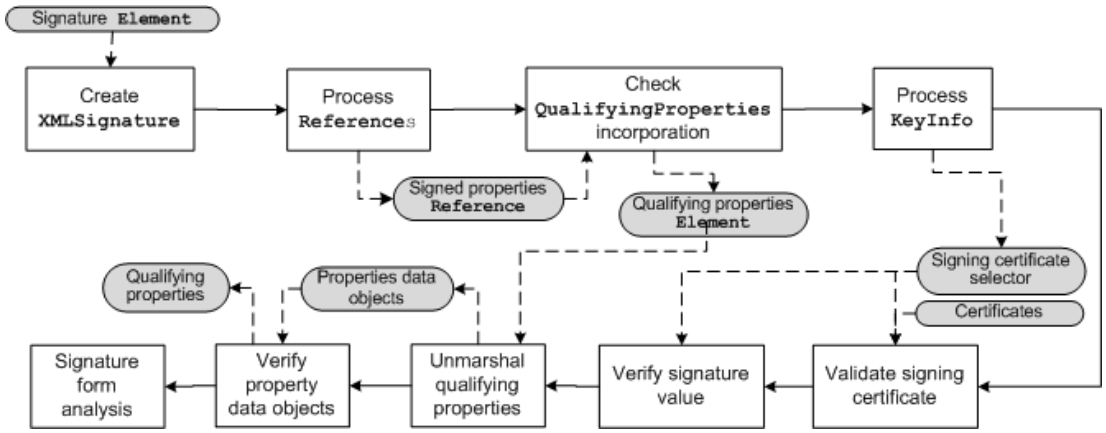

Figure 5.11: Signature verification steps

The following step is processing the `References` in order to identify the mandatory

`Reference` over the `SignedProperties` element. This includes checking if one and only one of such `Reference`s is actually present. If so, it has to be a same-document reference, because the library doesn't support the `QualifyingPropertiesReference` element. This means that the properties have to be on the signature's document, which is the common scenario [1].

Since all the `Reference`s are being iterated, this step is also an opportunity to create a representation of the signed data objects. This is done with the RawDataObjectDesc class, presented in Listing 5.19, which associates a Reference with a set of qualifying properties. The different instances of RawDataObjectDesc are part of the result of the verification process and in the end they will contain the data object properties that apply to the corresponding `Reference`.

Listing 5.19: The RawDataObjectDesc class

```
public class RawDataObjectDesc extends DataObjectDesc{
  public Reference getReference(){...}
  // ...
}
```

Going further on the verification, the correct incorporation of the qualifying properties has to be checked. This is done by analyzing the different `Object` elements in the signature and ensuring that only one `QualifyingProperties` element is present. In addition, the signed properties `Reference` from the previous stage is used to get the referenced `SignedProperties` element and ensure that it actually is a child of `QualifyingProperties`.

The next steps relate to the validation of the signing certificate, which may or may not be present in the signature but is always identified in the `SigningCertificate` property. Some additional information about the certificate may also be present in the `KeyInfo` element, including the certificate itself. The library assumes that `KeyInfo` contains at least one of the following elements that allow the selection of the signing certificate, in order of appearance:

1. Issuer name and certificate serial number.

2. Subject name.

3. A certificate.

[1] During project development the author didn't find any signatures using the `QualifyingPropertiesReference` element

Issuer/serial and subject name are preferred because multiple certificates may appear. If those elements are not present, the first certificate inside `KeyInfo` is assumed to be the signing certificate. Besides the signing certificate selection criteria, all the certificates found in `KeyInfo` are collected, as they may be needed to validate the leaf certificate. Then, the CertificateValidationProvider is invoked to validate the certificate, which in turn is used to verify the signature value (XML-DSIG verification).

After the core signature verification, the qualifying properties have to be verified. To accomplish this, the properties data is first unmarshalled from the DOM tree, resulting in the corresponding PDOs. Afterwards, each PDO is verified according to the property rules and the corresponding high-level objects are returned. These steps are the core of XAdES verification, hence being further discussed in the following sections. The qualifying properties are then analyzed to check if they correctly match one of the signature forms.

All the information gathered until this moment — validation data, signed data objects, qualifying properties, the signature form and the signature itself — is returned as a result of the verification. Before that, there is a final global verification, i.e. one that is done over the whole set of information resulting from the previous verifications. To that end, custom verifiers may be registered in the verification profile. The purpose of this last step is to allow the enforcement of rules that imply relations between properties and/or signature elements. The library actually uses one of these verifiers to check time-stamp coherence (for instance, time-stamps over references have to be prior to signature time-stamps).

5.3.3 Unmarshalling Qualifying Properties

The verification of qualifying properties is done over PDOs, which contain all the needed data. Therefore, before verifying the properties, their XML structure has to be processed in order to create the corresponding PDOs. The unmarshalling process is undertaken by a service provider that can be configured in the verification profile. It is represented by the QualifyingPropertiesUnmarshaller interface, shown in Listing 5.20.

Listing 5.20: The QualifyingPropertiesUnmarshaller interface

```
public interface QualifyingPropertiesUnmarshaller{
  void unmarshalProperties(Element qualifyingProps,
      QualifyingPropertiesDataCollector propertyDataCollector);
  void setAcceptUnknownProperties(boolean accept);
}
```

The unmarshaller's task is to process the `QualifyingProperties` element and create PDOs for **all** the existing properties. The data objects are supplied to the QualifyingPropertiesDataCollector, which controls the number of occurrences of the different data objects through multiple setters and a backing PropertiesSet instance.

The XAdES XML schema allows new property elements in the unsigned properties containers. However, the signature verifier may not intend to support those properties. This disposition is configured on the verification profile and will result on invoking the setAcceptUnknownProperties method of the unmarshaller. When unknown properties are supported they should be returned using a GenericDOMData instance.

As for other service providers there is a default implementation of the property unmarshaller, which relies on JAXB [32] to actually process the DOM tree. The process is similar to property marshalling, but in the opposite direction: the JAXB unmarshaller is used to create JAXB objects that represent the XML structure; then, a set of converters is used to create each PDO from the corresponding JAX object. These converters implement the QualifyingPropertyFromXmlConverter interface, shown in Listing 5.21.

Listing 5.21: The QualifyingPropertyFromXmlConverter interface

```
interface QualifyingPropertyFromXmlConverter<TXml>{
    void convertFromObjectTree(TXml xmlProps,
        QualifyingPropertiesDataCollector propertyDataCollector);
}
```

The TXml type parameter defines the JAXB class that corresponds to the immediate parent of the property being handled (`SignedSignatureProperties`, for instance). The converter uses the getters on the JAXB object to access the data for its type of property and creates the corresponding data object, which is then added to the property data collector.

5.3.4 Verifying Property Data Objects

The verification of property data objects is the core of XAdES verification; each property has a set of rules that has to be checked, ranging from simple conditions to digest comparison and time-stamp token verification. The principles for data object verification are very similar to the ones in data object generation and the same applies for the implementation strategy.

The verification of a PDO is based on the QualifyingPropertyVerifier interface, shown in Listing 5.22. Each verifier handles data objects for properties of a specific type. When

invoked, the verifier should check the appropriate rules and create the high-level property instance corresponding to the supplied data object, which will be part of the verification result. Any data related to the signature that is being verified, such as the signed data object representations, can be accessed through the QualifyingPropertyVerificationContext class.

Listing 5.22: The QualifyingPropertyVerifier interface

```
public interface QualifyingPropertyVerifier<TData extends PropertyDataObject>{
  public QualifyingProperty verify(TData propData,
      QualifyingPropertyVerificationContext ctx)
}
```

The verification process is triggered through an internal service provider, supplied to the XadesVerifierImpl instance, which starts by verifying the structure of the data objects. This is even more important during verification, as the signature structure may be incorrect. By validating the data objects structure at this point, one gets a solid basis for the upcoming procedures. The next step is to iterate the data objects and obtain a QualifyingPropertyVerifier for each of them. This is done through the QualifyingPropertyVerifiersMapper interface, presented in Listing 5.23, which provides the verifier suitable for a given data object.

Listing 5.23: The QualifyingPropertyVerifiersMapper interface

```
interface QualifyingPropertyVerifiersMapper{
    <TData extends PropertyDataObject> QualifyingPropertyVerifier<TData>
        getVerifier(TData p);
}
```

The purpose of this interface is to make the verification of data objects independent of how the individual verifiers are actually obtained. Similarly to data object generation, the implementation in use gets the verifiers from the verification profile. In addition, all the considerations made for dependencies of PDO generators apply to property verifiers.

5.4 Signature Enrichment

All the XAdES forms above XAdES-EPES are based on the inclusion of unsigned signature properties, which means they can be appended to the signature without breaking it. The XAdES specification takes this into account:

> "As a minimum, the signer will provide the XAdES-BES or when indicating that the signature conforms to an explicit signing policy the XAdES-EPES."
>
> "(...) The signer or a TSP could provide the XAdES-T. If the signer did not provide it, the verifier SHOULD create the XAdES-T on first receipt of an electronic signature (...)."
>
> "(...) when the signer does not provide the XAdES-C, the verifier SHOULD create the XAdES-C when the required components of revocation and validation data become available."

The *first receipt of an electronic signature* is likely to be the first verification of that signature. Thus, the verification process is the appropriate place to perform the *form extensions* described above. To that end, the XadesVerfier interface includes an additional method, as shown in Listing 5.24.

Listing 5.24: The XadesVerifier interface

```
public interface XadesVerifier{
  // ...
  XAdESVerificationResult verify(Element sigElem, XadesSignatureFormatExtender
      formExt, XAdESForm minForm);
}
```

The verification is done without any change but if a minimum form is not yet present, the signature is enriched using a XadesSignatureFormatExtender instance. This entity is responsible for adding a set of unsigned properties to an existing XMLSignature, as illustrated in Listing 5.25.

Listing 5.25: The XadesSignatureFormatExtender interface

```
public interface XadesSignatureFormatExtender{
  void enrichSignature(XMLSignature sig, UnsignedProperties props);
}
```

A XadesSignatureFormatExtender is easily implemented by reusing service providers from the signature production process, namely the PropertiesDataObjectsGenerator and the UnsignedPropertiesMarshaller. As for other signature services, there is a profile that allows its configuration. It is also worth pointing that UnsignedPropertiesMarshallers must support marshalling into a DOM tree that already contains property elements.

Although XadesSignatureFormatExtender handles a set of unsigned properties, some restrictions apply to those properties due to the library's processing model. Recall that in signature production the whole set of PDOs is generated before any of them is actually marshalled, which means that a property can't depend on another's XML structure. However, some qualifying properties that are mandatory in the extended forms have that kind of dependencies. This is the case of the `SigAndRefsTimeStamp` property which contains a time-stamp token calculated over `CompleteCertificateRefs`, among others. Consequently, these two properties can't be used simultaneously to extend the signature form. The practical consequence of this limitation is that some form transitions are not possible. For instance, if one has a XAdES-BES signature it isn't possible to directly extend its form to XAdES-X because this would involve the two properties described above. Instead, the signature might be extended to XAdES-C and then to XAdES-X.

Some properties on the extended forms also take into account the XML elements ordering. For instance, the `SigAndRefsTimeStamp` property is calculated only over the previous eligible siblings. The library's processing model doesn't take into account the element ordering, which means that signatures with extended forms cannot be verified.

5.5 Profile Resolution

The signature services made available by the library are based on profiles, which are configured in order to obtain instances of types that actually implement those services (SignerBES, for instance). The service providers that are necessary for the operation of the services are supplied by the profile when a service is requested. However, not all the providers need to be configured by the developer because the library has default implementations for many of them, which are used if the configuration is not overridden.

The same way signature services implementations depend on a set of service providers, the providers themselves may have dependencies. When a service instance is requested to a profile all these chained dependencies have to be resolved. This is done with *dependency injection*, more precisely with Google's dependency injector: *Guice* [33] (Appendix B contains a brief introduction to Guice). It was chosen for two main reasons: first, the interface for binding definitions — associations between interfaces and instances or classes — is very expressive; second, it has support for bindings with generic types. The only drawback is that the different classes have to use the @Inject annotation on their setters and constructors in order to have their dependencies injected.

During profile configuration, service providers may be registered by instance or by type. When registered by type, a provider may have other dependencies, which should be flagged with the @Inject annotation. Those dependencies may refer to components in the library or any other types, as the profiles support general purpose bindings, i.e., mappings for dependencies other than the built-in service providers.

All the different types of profiles rely on the XadesProfileCore class, which actually implements the bindings registration and the dependency resolution using Guice. The class has a set of methods for registering dependency mappings, including mappings for generic types. These bindings will be used to create a dependency container (the *injector*) that is then used to create instances of the requested services. In Guice, the bindings are gathered in *modules*, which serve as parameters for injector construction. However, the module is only invoked to configure its bindings during the creation of the injector. Therefore, the bindings configured on XadesProfileCore are internally stored using a BindingAction, as illustrated in Listing 5.26.

Listing 5.26: Storing profile bindings

```
public final class XadesProfileCore{
  public <T> void addBinding(Class<T> from, Class<? extends T> to){
        this.bindings.add(new BindingAction(){
            @Override
            public void bind(Binder b){
                b.bind(from).to(to);
            }});
  }
  // ...
}
```

When an instance of a service is requested, the existing bindings are used to create a module which in turn is used to create a new injector, as shown in Listing 5.27. Actually, the module is used to override the configurations of another module containing the default bindings, which is supplied by each type of profile.

A new injector is created every time the getInstance method is invoked to ensure that the resulting instance always reflects the current profile configuration.

The usage of dependency injection goes beyond the direct creation of signature services instances. Recall the PropertyDataGeneratorsMapper interface, used during signature production to find property data object generators. The implementation of this internal service has a

Listing 5.27: Using the bindings to create service instances

```
public <T> T getInstance(Class<T> clazz, Module defaultsMod){
  Module usrMod = new Module(){
      @Override
      public void configure(Binder b){
          for (BindingAction ba : bindings)
              ba.bind(b);
      }};
  Module finalMod = Modules.override(defaultsMod).with(usrMod);
  return Guice.createInjector(finalMod).getInstance(clazz);
```

dependency on the injector itself, as shown in Listing 5.28. The injector is used to obtain the data object generators, which are configured as part of the signature profile default bindings. In addition, custom generators may be registered through the profile, becoming available in the injector.

Listing 5.28: Obtaining PropertyDataObjectGenerator instances

```
class PropertyDataGeneratorsMapperImpl implements ...{
  @Inject
  public PropertyDataGeneratorsMapperImpl(Injector injector){...}
  @Override
  public PropertyDataObjectGenerator<TProp> getGenerator(TProp p){
    ParameterizedType pt =
        Types.newParameterizedType(PropertyDataObjectGenerator.class,
        p.getClass());
    return injector.getInstance(Key.get(TypeLiteral.get(pt)));
  }
}
```

Note that the data object generators are requested by type, which means that the injector will also handle their dependencies in order to return the requested instance. A similar strategy is used for property verifiers and the verification profile.

5.6 Exception Model

XAdES4j has an hierarchic exception model on which all the exceptions are ultimately based on XAdES4jException. The exception hierarchy follows a functional division, i.e., the excep-

tions are grouped according to the functionality that may throw them. Examples of this separation in the first level of the hierarchy are:

- KeyingDataException — indicates that the keying data for signature production cannot be obtained.
- MarshalException — represents an error during the marshalling of property data objects.
- PropertyDataGenerationException — indicates an error during the generation of property data objects.
- InvalidSignatureException — indicates that XML-DSIG and/or XAdES verification rules are not fulfilled.

The branches in the hierarchy are subdivided into more specific exceptions which may be "leaf" exceptions — representing a specific failure — or may themselves start another branch, as illustrated in Figure 5.12. This enables the developer to go from rough to fine grained exception filtering by pointing at different levels in the hierarchy.

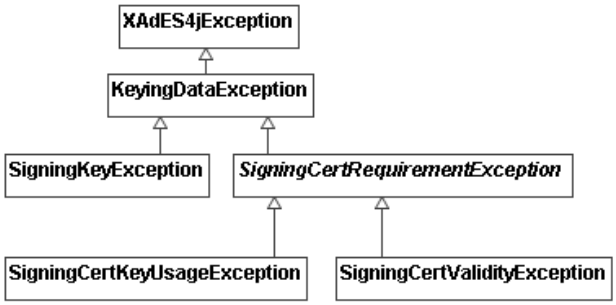

Figure 5.12: Excerpt of the exception hierarchy

The exception hierarchy goes down to a considerable detail so that each exception represents a specific failure (refer to Appendix C for further detail). This helps the developer narrowing the errors and reacting accordingly. However, additional information about the failure may be useful, specially because one doesn't know how the exceptions will be handled. For instance, if a certificate doesn't include the appropriate key usage the corresponding exception should include the certificate itself. If a simple error message is sufficient, the exception will provide it; if a more specific handling is required, the exception also includes the needed information. One can think of exceptions as data models of the errors.

This principle is widely used in the library, specially in the exceptions related to the verification of qualifying properties (exceptions ensure that rule violations don't go unno-

ticed). As exemplified in Figure 5.13, when a certificate reference in the `SigningCertificate` property cannot be verified due to digests mismatch, the SigningCertificateReferenceException contains the certificate and the reference itself. Likewise, when the object reference on a `DataObjectFormat` property cannot be resolved, the resulting DataObjectFormatReferenceException contains that reference.

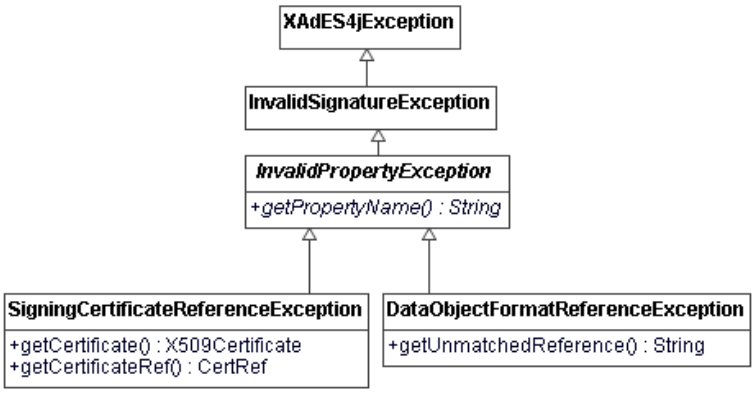

Figure 5.13: Exceptions during properties verification

5.7 Bundled Service Providers

As previously discussed, the library includes default implementations for most of the service providers involved in the signature services. However, some providers, such as the KeyingDataProvider, can't have default implementations. Nevertheless, the library includes implementations for common use cases of such providers, three of them described in the following sections.

5.7.1 Keying Data Provider

A common approach to access keying data in the Java platform is using the KeyStore engine class. It allows access not only to file-system and Windows key stores, but also to cryptographic tokens using a PKCS#11 provider. *XAdES4j* includes the KeyStoreKeyingDataProvider class, which is a KeyingDataProvider based on a KeyStore. In order to access the key store, two helper interfaces are used: KeyEntryPasswordProvider and SigningCertSelector, shown in Listing 5.29.

The first interface is used to get the password for accessing a key entry, while the second selects the signing certificate among the certificates in the key store. Implementations of these interfaces have to be supplied when the KeyStoreKeyingDataProvider is created.

Listing 5.29: The KeyEntryPasswordProvider and SigningCertSelector interfaces

```
public interface KeyEntryPasswordProvider{
  char[] getPassword(String entryAlias, X509Certificate entryCert);
}
public interface SigningCertSelector{
  X509Certificate selectCertificate(List<X509Certificate> certs);
}
```

To obtain the signing certificate chain, the following actions are taken:

1. Iterate the key store entries; for each private key entry add its certificate to a list.
2. Invoke the SigningCertSelector passing the list from the previous step.
3. Access the entry corresponding to the certificate resulting from the previous step and get the associated certificate chain.

To access the private key matching the signing certificate the procedure is as follows:

1. Get the alias for the entry matching the given certificate.
2. Create a ProtectionParameter using the KeyEntryPasswordProvider.
3. Access the key store entry using the protection resulting from the previous step and get its private key.

The KeyStoreKeyingDataProvider class is actually abstract. The library provides two subclasses, one for accessing file-system key stores and another for accessing cryptographic tokens using SUN's PKCS11 provider. The main difference is in obtaining the KeyStore instance: on the first case the key store type and physical path have to be supplied; on the other, only the location of the PKCS11 native library is needed. In addition, the ProtectionParameter used to get the private key is also different: for file-system key stores, a PasswordProtection is used; on the other hand, for PKCS11 key stores a CallbackHandlerProtection is used, which delays the call to the KeyEntryPasswordProvider. Actually, the protection for PKCS#11 key stores may be omitted if the native library controls the access on its own (this is the case of the Portuguese Citizen Card middleware).

5.7.2 Time-stamp Verification

A TimeStampVerificationProvider is used during signature verification whenever a time-stamp token property is being verified. As illustrated on Listing 5.30, the purpose of this service provider is to verify an encoded time-stamp token, which includes checking the embedded digest value against the actual digest of the input for that time-stamp (the input depends on the property being verified).

Listing 5.30: The TimeStampVerificationProvider interface

```
public interface TimeStampVerificationProvider{
  Date verifyToken(byte[] token, byte[] tsInput);
}
```

The library includes a default implementation for this service, based on SUN's proprietary API on the sun.security.pkcs package, which includes types that can parse the tokens and provide access to its members.

Verifying a time-stamp token includes two main tasks: verifying the signature and checking the digest value. The last step is straightforward: the input is digested using the algorithm that is specified in the token and the resulting value is compared to the message imprint also contained in the token. Checking the signature value is also simple because the token includes the TSA's certificate and the types on SUN's API already provide that functionality. Note that the signature is not a XML signature, because the time-stamp token is a CMS [34] `SignedData` structure. In order to complete the token verification a final step is needed: validate the TSA's certificate. Recall that the library already defines an interface for a service provider that validates certificates: CertificateValidationProvider. Thus, this becomes a dependency of the TimeStampVerificationProvider implementation, as illustrated in Listing 5.31.

This service provider is an example of how easy it is to rely on existing functionality, as a consequence of the profile resolution strategy. Note that there are property data object verifiers that have a dependency on TimeStampVerificationProvider. When any of those data object verifiers is requested during signature verification, that dependency may be resolved to the default implementation, which in turn has its own dependency on CertificateValidationProvider.

Listing 5.31: The default TimeStampVerificationProvider

```
public class DefaultTimeStampVerificationProvider implements
    TimeStampVerificationProvider{
  @Inject
  public DefaultTimeStampVerificationProvider(CertificateValidationProvider
      certificateValidationProvider){...}
  // ...
}
```

5.7.3 Certificates Validation

One of the important actions undertaken during signature verification is validating certificates, namely the signer's and the ones from TSAs. As previously discussed, this task is accomplished through a CertificateValidationProvider, illustrated in Listing 5.32 for an easier reading.

Listing 5.32: The CertificateValidationProvider interface

```
public interface CertificateValidationProvider{
  ValidationData validate(X509CertSelector certSelector,
      Collection<X509Certificate> otherCerts);
}
```

The library includes an implementation of this service that relies on the commonly used Java Certification Path API, namely on the classes for building and validating X.509 certification paths according to the PKIX algorithm. As illustrated in Listing 5.33, this service provider requires a KeyStore — which contains trust-anchor certificates — and a set of CertStores containing certificates and CRLs to be used in the validation process.

Listing 5.33: The PKIX certificate validator

```
public class PKIXCertificateValidationProvider implements
    CertificateValidationProvider{
  public PKIXCertificateValidationProvider(KeyStore trustAnchors, boolean
      revocationEnabled, CertStore... intermCertsAndCrls){...}
  // ...
}
```

When a validation is requested the parameters for the PKIX CertPathBuilder are obtained

as follows:

1. Create the instance by supplying the trust-anchors key store on the provider and the certificate selector received in the validate method.

2. Add the certificates supplied in the validate method. This is done using a *collection* CertStore.

3. Add the CertStores on the provider.

4. Set revocation usage as specified in the provider.

The certification path returned by the CertPathBuilder doesn't include the trust-anchor certificate. However, the resulting validation data should contain the full certification path. To that end, the PKIX-specific builder result is used (PKIXCertPathBuilderResult), which provides access to the certificate of the trust-anchor in the path.

If revocation is enabled the CRLs used in the validation also have to be returned. Even though they were included in the construction of the certification path through the provider's CertStores, the builder doesn't provide access to the CRLs that were actually used. Therefore, the last step is to determine which CRLs were used to check the status of the different certificates in the chain. Note that one needs a valid CRL for each issuer in the path. The search is done with the following steps:

1. Map each issuer's name to its certificate.

2. Create a X509CRLSelector with issuer names and the current date as selection criteria.

3. Select CRLs from the different CertStores in the provider using the filter just created and collect them using a Set.

4. Verify the signature on each CRL using the issuer's certificate obtained from the mapping previously created. Note that the certificate itself was already validated as part of certification path construction.

5. Return the collected CRLs.

The result of certificate validation contains both the certification path and the CRLs used to check the statuses of the different certificates.

5.8 Tests

To validate the library's features a set of test cases were developed. Some of them are unit tests that cover specific items in the library but the majority are tests over the signature production and verification processes as a whole. The library is not only tested on correct usage scenarios — by producing and validating signatures that are well-formed and valid — but also in failure situations to ensure the correct behavior when processing invalid and badly formatted signatures.

The library test cases are categorized as follows:

1. Self production and verification — a set of signatures is both produced and verified by the library. The tests include the generation of signatures in all the main XAdES forms and with all the optional signature properties, using the different signature profiles and XadesSigners.

2. Third party production and self verification — a set of signature produced by third parties is verified by the library. This is important to ensure interoperability. The tests include the verification of XAdES signatures produced by the *jXAdES* library and by Member States of the European Union (unknown production context).

3. Failure situations — multiple valid signatures are deliberately tampered and their verification is attempted. The main purpose is validating the exception model, i.e. assuring that the appropriate exception is thrown on a specific failure scenario. The tests include errors such as incorrect incorporation of qualifying properties, invalid data object references, invalid signature value, missing certificates and invalid properties.

Another important group of tests would encompass the verification of signatures produced by *XAdES4j* using a third party XAdES implementation. However, none of the analyzed libraries throughly supports XAdES verification. Appendix D summarizes the most relevant test cases for each category.

The different signatures are produced and verified using the following PKI data:

- Key and certificate chain generated for test purposes using the *makecert* tool.

- A set of keys, certificates and CRLs from the National Institute of Standards and Technology (NIST) PKI test suite [35], namely in XAdES-C signatures.

- The keys and certificates on the Portuguese Citizen Card and the corresponding certification path.
- Certificates and CRLs for the certification authorities of the TSAs.

When the signatures include time-stamp properties, different TSAs are used to obtain the respective tokens. Finally, the tests also cover the usage of service providers other than the defaults.

CHAPTER 6

Conclusions

This document describes the design and implementation of a Java library for XML Advanced Electronic Signatures services, which was named *XAdES4j*. The library is compliant with the XAdES version 1.4.1 specification and supports the production and verification of signatures in the four main XAdES forms: XAdES-BES, XAdES-EPES, XAdES-T and XAdES-C. In addition, the extended forms are supported by the addition of the corresponding qualifying properties to an existing signature.

The initial study allowed the author to become aware of the serious efforts that the European Union has been undertaking to establish solid bases for electronic signatures and to promote their acceptance, not only by providing legal foundations but also standard formats, equipment requirements, and multiple implementation guidelines. Also, it became clear how the lack of qualifying information in XML-DSIG makes it insufficient in that context and how XAdES overcomes that issue. The most important features are long term validity and non-repudiation, supported by the identification of the signing certificate, the incorporation of validation data and the use of time-stamps.

Despite the growing importance of XAdES, the Java platform currently has no support for those signatures: only XML-DSIG is supported. The motivation for this project emerges directly from that fact.

The XAdES specification defines signature forms and processing rules but those are only the starting point when developing a solution for XAdES services. Several other questions arise, most of them related to how to actually gather the information that makes up the

signature. For instance, one needs to know how to obtain the signing key, how to validate a certificate and how to obtain a time-stamp token. All these aspects have to be accounted when designing the solution.

The *XAdES4j* library is an high-level, configurable and extensible solution for XAdES signature services. Its API is designed to abstract the developer from the signature XML structure and processing rules, which are handled internally. This includes handling all the detailed property contents which often involve digest calculations, interaction with TSAs and associations between XML elements.

The library is designed around two main concepts: profiles and service providers. A profile is a set of invariant characteristics that are used on a signature service, such as the signing key/certificate, the set of optional signature properties, the XAdES forms or the validation data. Instead of directly depending on those characteristics, the profiles are based on service providers which represent means of obtaining the needed information and/or functionality. This includes selecting the signing key, validating certificates, interacting with TSAs and marshalling/unmarshalling XML. Service providers are plugged into the library through well defined interfaces and can be configured independently without altering the core mechanisms. With this approach, the library behaves as a framework in the core of signature production and verification, as it will invoke the different service providers as needed. Also, the library is complete, meaning it includes implementations for the different service providers, some of them used by default.

The core XML-DSIG structure is created and processed using Apache XML Security, which is an open implementation of that standard. Besides the proper assembling of XML-DSIG elements through the Apache API, when producing a signature the library has to generate the qualifying properties XML elements. This is done in two steps: the first is to generate property data objects (PDO) containing the low-level property data, i.e., all the data that is needed to the final XML structure; the second step is marshalling the PDOs to obtain the properties DOM tree. Signature verification encompasses opposite actions: the PDOs are unmarshalled from the signature's XML and submitted to property verification, which results on the high-level property representations. New qualifying properties may be included by defining their high-level types and the corresponding PDOs, as well as the appropriate PDO generators and verifiers.

The initial objectives for the library were met: it enables producing, verifying and extending signatures in the main XAdES forms. The optional goals were partially met, as the

library only supports the extended XAdES forms through signature enrichment.

The library's features were validated through a set of tests that encompass both correct and incorrect use-cases. The tests for signature production include producing signatures in all the main forms, which are then used as input for the tests of the signature verification process. Besides verifying signatures that are known to be valid, the tests also cover signatures that are intentionally bad formed or invalid in order to ensure the library's correct behavior on failure scenarios. Finally, as an interoperability test, the library successfully verified third party signatures produced by Member States of the EU and by the *jXAdES* library.

6.1 Comparison with Existing Solutions

Generally, *XAdES4j* offers more features than any of the evaluated solutions, as none of them enables the production and verification of signatures in all the main XAdES forms.

Among the evaluated solutions, the *WebSign* XAdES module is the one that offers less functionality. It is a Java mirror of the XAdES properties' structure, handling their marshalling and unmarshalling. As such, it can't be reasonably compared to *XAdES4j*, which is at an higher level of abstraction and functionality. Nevertheless, if it is being used, the developer has to handle the generation of all the low-level property data, which doesn't happen with *XAdES4j*.

The *jXAdES* project and *XAdES4j* have similar principles regarding the core XML-DSIG structure and the incorporation of qualifying properties: an existing XML-DSIG implementation is used without exposing those details to the developer. With *jXAdES* one is able to produce signatures in all the main forms, but in some cases the mandatory properties are not automatically added. On the other hand, *XAdES4j* enforces the presence of the mandatory properties for the different forms. The definition of data object properties is also different: using *jXAdES* they are defined separately from the data objects themselves — because there is no representation of a signed data object — and the developer is responsible for establishing the corresponding association. With *XAdES4j* the properties are defined over data object representations and the structural association is created internally.

When it comes to identifying the signed resources, *jDigiDoc* and *XAdES4j* have similar principles: they use high-level representations of signed data objects that will result on the appropriate `Reference` elements, even though *jDigiDoc* is limited to the *DigiDoc* format. Another similar principle is the abstraction from the generation of the low-level property data and XML structure. However, *jDigiDoc* only supports the qualifying properties that are

part of the *DigiDoc* XAdES profile, while *XAdES4j* supports most of the qualifying properties.

As a final note, none of the analyzed libraries has a concept that matches the *XAdES4j* service providers or the independent configuration of components in signature operations. For instance, *jXAdES* depends on a specific means of obtaining time-stamp tokens. Furthermore, none of them is as configurable and extensible as *XAdES4j*.

6.2 Critical Analysis and Future Work

Although the initial objectives have been met, the library has some limitations and possible improvements. Both signature production and verification only support qualifying properties in the signature's document. In other words, the `QualifyingPropertiesReference` element is not supported. This is not very restrictive as none of the tested third party signatures includes that element. Furthermore, both the *jXAdES* and *jDigiDoc* projects use same-document properties. Using the `QualifyingPropertiesReference` element would also make it harder to check the proper incorporation of qualifying properties because multiple documents and `QualifyingProperties` elements would be involved.

A more significant limitation is that the library doesn't support OCSP validation data (only CRLs are supported). Using OCSP validation is practical because one checks the status of a specific certificate on demand. This results in less data and no need to keep a CRL repository up to date. Nevertheless, the XAdES specification only requires support for one of the validation data types. OCSP support was not included due to time restrictions, as the author would need to become familiar with the protocol and an API to support the actual communications. Implementing OCSP support would involve defining a new service provider to obtain OCSP responses and adjusting the existing validation data properties' types and the corresponding generation/verification processes.

The library's support for the extended XAdES forms is limited to signature enrichment, i.e., adding extra qualifying properties to an existing signature. Not being able to directly generate signatures with extended forms is not a significant restriction because those forms involve data used in signature validation, which is likely to be collected by the verifier. However, this limitation comes from the library's design. During signature production the whole set of PDOs is generated before any of them is actually marshalled, which means that a property can't depend on another's XML structure. This is not a problem in the main forms, but some qualifying properties that are mandatory in the extended forms have that kind of dependencies. This means that they can't be simultaneously present when generating a

signature, hence creating the limitation. In addition, some properties on the extended forms also take into account the XML elements ordering. For instance, the `SigAndRefsTimeStamp` property is calculated only over the previous eligible siblings. The library's processing model doesn't take into account the PDOs ordering when verifying a signature, which means that extended forms cannot be verified. Resolving the extended forms limitation on both signature production and verification would involve individual PDO marshalling — so that the DOM tree is progressively created — and ensuring that the PDOs are processed in DOM tree order.

Due to the extended forms limitation, the signatures that were produced using those forms for test purposes were not verified. In addition, none of the produced signatures was externally verified, because that operation is not throughly supported by the analyzed libraries. Verification by third parties would provide an important assurance of interoperability. The verification of signed data object properties is also not throughly tested because third party signatures containing those properties couldn't be found.

Finally, it could be useful to have a non-programmatic way to configure the default service providers, such as a configuration file. Note that one may override the defaults for a specific profile by subclassing its class and always using the new type, but that won't be a profile-wide configuration.

6.3 Final Notes

The *XAdES4j* library has been published under the GNU General Public License (GPL) at http://xades4j.googlecode.com.

References

[1] W3C, *Extensible Markup Language (XML) 1.1 (Second Edition)*, August 2006.

[2] W3C, *Document Object Model Level 2 Core Specification*, November 2000. http://www.w3.org/TR/DOM-Level-2-Core/.

[3] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon, *XML Signature Syntax and Processing (Second Edition)*, June 2008.

[4] ETSI, *XML Advanced Electronic Signatures (XAdES) 1.4.1*, June 2009. TS 101 903.

[5] C. Adams, P. Cain, D. Pinkas, and R. Zuccherato, *Internet X.509 Public Key Infrastructure Time Stamp Protocol (TSP)*. IETF RFC 3161.

[6] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, May 2008. IETF RFC 5280.

[7] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, *X.509 Internet Public Key Infrastructure Online Certificate Status Protocol — OCSP*, 1999. IETF RFC 2560.

[8] E. Parliament, "Directive 1999/93/ec on a community framework for electronic signatures," *Official Journal of the European Communities*, 1999.

[9] M. of Economy Trade and I. of Japan, *Long term signature profiles for XAdES*, 2008. http://www.jipdec.or.jp/archives/ecpc/longtermstorage/en/index.html.

[10] C. G. da Infra-estrutura de Chaves Públicas Brasileira, *Perfil para Assinaturas XAdES na ICP-Brasil*, 2008. http://www.serpro.gov.br/serpronamidia/2008/dezembro/padroes-de-assinatura-digital-aprovados-pelo-comite-gestor.

[11] S. Gu, "Digital signatures in office 2010," *Microsoft Office 2010 Engineering (August 2010)*. http://blogs.technet.com/b/office2010/.

[12] A. X. Project, *Apache XML Security*. http://santuario.apache.org/.

[13] *XML Digital Signature, Cover Pages technology report*. http://xml.coverpages.org/xmlSig.html (May 2010).

[14] OASIS, *Web Services Security*, 2006.

[15] *XPointer Framework*, March 2003.

[16] J. Boyer and G. Marcy, *Canonical XML Version 1.1*, May 2008.

[17] J. Boyer, D. E. Eastlake, and J. Reagle, *Exclusive XML Canonicalization Version 1.0*, July 2002.

[18] C. of the European Communities, "Report on the operation of directive 1999/93/ec on a community framework for electronic signatures," March 2006.

[19] C. of the European Communities, "Standardisation mandate in the field of information and communication technologies applied to electronic signatures, m/460," December 2009.

[20] ETSI, *Electronic Signatures and Infrastructures (ESI); CMS Advanced Electronic Signatures (CAdES)*, November 2009. TS 101 733.

[21] ETSI, *TC Security - Electronic Signatures and Infrastructures (ESI); XML format for signature policies*. TS 102 038.

[22] *Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks*. ITU-T Recommendation X.509.

[23] "Java xml digital signature api." Java Specification Request 105, http://www.jcp.org/en/jsr/detail?id=105.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[25] S. Microsystems, "Java standard edition source code." org.jcp.xml.dsig.internal.dom.DOMXMLObject class.

[26] T. Imamura, B. Dillaway, and E. Simon, *XML Encryption Syntax and Processing*, December 2002.

[27] M. Nachev, "jxades — java implementation of xades." http://xades.dev.java.net/ (August 2010).

[28] "As sertifitseerimiskeskus (sk) — certification center." http://www.sk.ee/ (August 2010).

[29] "Openxades — jdigidoc." http://www.openxades.org/ (August 2010).

[30] A. S. (SK), *DigiDoc Format Specification v1.3*, May 2004.

[31] R. Cardon, "Websign project — xades module." http://rcardon.free.fr/websign/ (August 2010).

[32] *Java Architecture for XML Binding — Reference Implementation Project.* http://jaxb.dev.java.net/.

[33] Google, *Guice — a lighweight dependency injection framework for Java.* http://code.google.com/p/google-guice/.

[34] R. Housley, *Cryptographic Message Syntax*, July 2004. IETF RFC 3852.

[35] N. I. of Standards & Technology (NIST), *X.509 Path Validation Test Suite.* http://csrc.nist.gov/groups/ST/crypto_apps_infra/pki/pkitesting.html.

APPENDIX A

Package Organization

Table A.1 summarizes the Java packages that make up *XAdES4j*. For each package there is a brief description and a reference to the most relevant classes. Exception classes are not referred, but each package contains the ones that relate to its features.

Package	**Description**
xades4j	The root package.
xades4j.production	Support for signature production. Contains the XadesSigningProfile class and its subclasses, as well as the XadesSigner interface and its realizations. Furthermore, it contains the classes related to property data object generation and signed data objects descriptions.
xades4j.properties	Contains the types that represent the different XAdES qualifying properties, such as QualifyingProperty, SignedDataObjectProperty and SignatureTimeStampProperty.
xades4j.properties.data	Definition of the different property data objects and support for structural verification. Contains the PropertyDataObject and PropertyDataObjectStructureVerifier interfaces.
xades4j.providers	Interfaces for the different service providers, such as KeyingDataProvider, CertificateValidationProvider and TimeStampTokenProvider.

Package	Description
xades4j.providers.impl	Implementations (default and other) of the different providers.
xades4j.utils	Helper classes used in the library.
xades4j.verification	Support for signature verification. Contains the XadesVerificationProfile class and the XadesVerifier interface. Furthermore, it contains the classes related to the verification of individual qualifying properties.
xades4j.xml.bind	Auto-generated classes that support JAXB marshalling/unmarshalling, used by the default.
xades4j.xml.marshalling	Contains the types related to marshalling the property data objects, namely the PropertiesMarshaller and QualifyingPropertyDataToXmlConverter interfaces.
xades4j.xml.unmarshalling	Contains the types related to unmarshalling the property data objects, namely the QualifyingPropertiesUnmarshaller and QualifyingPropertyFromXmlConverter interfaces.

Table A.1: XAdES4j package organization

APPENDIX B

Dependency Injection with Guice

Dependency injection is a type of inversion of control where the concept being inverted is the process of obtaining a needed component. When a type has a dependency on another, instead of directly creating new instances it has them supplied through its constructor (they are *injected*). There are other types of dependency injection, namely through setters, but constructor injection is preferred. The dependencies are expressed as interfaces, which not only makes a type independent from how its dependencies are created but also from their actual implementation.

Having the types designed in consonance with this pattern, the main issue is resolving the dependencies when creating new instances. There are several frameworks that can be used to accomplish that task. These frameworks are usually referred to as *dependency containers* or *injectors* because it is their responsibility to analyze a type's dependencies and provider the appropriate components. Guice is one of those frameworks and provides support for identifying, configuring and resolving dependencies.

The first step is to identify the dependencies of the types being defined and code them as constructor parameters. Then, one indicates that those dependencies should be injected using the `@Inject` annotation. Listing B.1 illustrates this scenario using the property data object generator for the `SignatureTimeStamp` property, which requires a `TimeStampTokenProvider`.

Listing B.1: Identifying dependencies

```
class DataGenSigTimeStamp implements ... {
```

```
  @Inject
  public DataGenSigTimeStamp(TimeStampTokenProvider
      timeStampTokenProvider){...}
}
```

The next step is providing Guice the needed information to resolve the dependency, i.e., configuring the class of the time-stmap provider that will actually be injected. This is done with a *binding*, defined over the Binder class, as illustrated in Listing B.2. A binding is an association between a type (an interface) that is used to express dependencies and a means of resolving that dependency. The most common ways of resolving a dependency is by binding it to an instance or to a type whose instances will be created by the injector as needed.

Listing B.2: Configuring dependencies

```
Binder b = ...;
b.bind(TimeStampTokenProvider.class).to(DefaultTimeStampTokenProvider.class);
b.bind(DataObjectPropertiesProvider.class).toInstance(new
    DataObjectPropertiesProvider(){
  @Override
  public void provideProperties(DataObjectDesc dataObj){
      // By default no properties are specified for a data object.
  }
});
```

Note that if one is binding to classes, the dependencies of the resolving types are also processed when new instances are needed, and the same happens for its dependencies; this process goes on until the needed object graph can be created.

Bindings are grouped in *modules*, as illustrated in Listing B.3. One or more modules are used to create an Injector, which can then be used to obtain instances of types that have their dependencies resolved according to the modules' bindings.

Listing B.3: Creating a Module and configuring the Injector

```
Module m = new Module(){
  @Override
  public void configure(Binder b){
    b.bind(TimeStampTokenProvider.class).to(DefaultTimeStampTokenProvider.class);
  }};
Injector i = Guice.createInjector(m);
DataGenSigTimeStamp gen = i.getInstance(DataGenSigTimeStamp.class);
```

APPENDIX C

Exception Model

The exception model is organized as follows. Each indentation step indicates that the following classes are subclasses of the previous one.

- XAdES4jException
  - UnsupportedAlgorithmException
  - KeyingDataException
    - SigningKeyException
    - SigningCertChainException
    - SigningCertRequirementException
      - SigningCertKeyUsageException
      - SigningCertValidityException
  - PropertyDataGenerationException
    - PropertyDataGeneratorErrorException
    - PropertyDataGeneratorNotAvailableException
  - PropertyDataStructureException
    - PropertyDataStructureVerifierNotAvailableException
  - MarshalException
    - UnsupportedDataObjectException
  - UnmarshalException
    - PropertyUnmarshalException

- InvalidSignatureException
  - CoreVerificationException
    - ReferenceValueException
    - SignatureValueException
  - QualifyingPropertiesIncorporationException
  - QualifyingPropertyVerifierNotAvailableException
  - InvalidPropertyException
    - PropertyVerifierErrorException
    - (Property-specific exceptions)
  - InvalidXAdESFormException
- CertificateValidationException
  - CannotSelectCertificateException
  - CannotBuildCertificationPathException
  - InvalidKeyInfoDataException
- TimeStampTokenGenerationException
- TimeStampTokenVerificationException
  - TimeStampTokenStructureException
  - TimeStampTokenSignatureException
  - TimeStampTokenDigestException
- XAdES4jXMLSigException

APPENDIX D

Test Cases

This appendix summarizes the most relevant test cases implemented to validate the library's features. The code and resources for the tests can be found under the `src\test` folder on the library's source.

D.1 Signature Production

Table D.1 describes the tests cases regarding signature production. The resulting signatures are used as input for the subsequent tests of the signature verification process.

Test Name	**Description**
SignBES	Produces a XAdES-BES signature. Includes all the signed data object properties.
SignBESExtrnlRes	Produces a XAdES-BES signature over a resource on the Web.
SignBESWithCounterSig	Produces a XAdES-BES signature using a custom SignaturePropertiesProvider which includes the `CounterSignature` unsigned property.
SignEPES	Produces a XAdES-EPES signature using a simple string as the policy document.
SignTWithPolicy	Produces a XAdES-T signature based on XAdES-EPES with implied policy.

Test Name	Description
SignTPtCC	Produces a XAdES-T signature based on XAdES-BES using the Portuguese Citizen Card through the PKCS#11 provider (Windows only).
SignC	Produces a XAdES-C signature using the PKI data from NIST [35].

Table D.1: XAdES4j test cases for signature production

D.2 Verification of XAdES4j Signatures

Table D.2 describes the tests cases regarding signature verification, using signatures that were produced by the library.

Test Name	Description
VerifyBESCounterSig	Verifies a XAdES-BES containing the `CounterSignature` unsigned property.
VerifyBESEnrichT	Verifies a XAdES-BES signature and extends its format to XAdES-T upon verification.
VerifyBESExtrnlResEnrichC	Verifies a XAdES-BES signature calculated over a resource on the Web and extends its format to XAdES-C.
VerifyTEPES	Verifies a XAdES-T signature based on XAdES-EPES.
VerifyTPTCC	Verifies a XAdES-T signature produced using the Portuguese Citizen Card (Windows only).
VerifyCEnrichXL	Verifies a XAdES-C signature and extends its format to XAdES-X-L.
VerifyBESCustomPropVer	Verifies a XAdES-BES signature using a custom verifier for the `SigningTime` signed property.

Table D.2: XAdES4j test cases for signature verification

D.3 Verification of Third Party Signatures

Table D.3 describes the tests cases regarding signature verification, using third party signatures. Most of the signatures are part of Trust Service Provider Lists (TSLs) produced by Member States of the European Union.

Test Name	**Description**
VerifyPTTSL	Verifies a XAdES-BES signature produced by the Portuguese Government.
VerifyESTSL	Verifies a XAdES-BES signature produced by the Spanish Government.
VerifyBETSL	Verifies a XAdES-BES signature produced by the Belgian Government.
VerifySKTSL	Verifies a XAdES-BES signature produced by the Government of Slovakia. This signature includes some optional signature and data object properties.
VerifyPetition	Verifies a XAdES-T signature produced with the *jXAdES* library.

Table D.3: XAdES4j test cases for verification of third party signatures

D.4 Error Scenarios

Table D.4 describes the test cases over signatures that were intentionally altered to be invalid.

Test Name	**Description**
ErrVerifySignedPropsIncorp	Tests the detection of a `SignedProperties` element that is not properly incorporated.
ErrVerifyIncorrectC	Tests the detection of a missing `CompleteRevocationRefs` property on a XAdES-C signature.
ErrVerifyNoSignCert	Tests the impossibility of verifying a signature that doesn't contain the signing certificate.
ErrVerifyChangedSigValue	Tests the detection of an invalid signature value.
ErrVerifyCMissingCertRef	Tests the detection of a missing CA certificate reference on a XAdES-C signature

Test Name	**Description**
ErrVerifyUnmatchSigTSDigest	Tests the detection of a `SignatureTimeStamp` whose digest doesn't match the actual digest of `SignatureValue`.

Table D.4: XAdES4j error test cases