Experiences with the Enforcement of Access Rights Extracted from ODRL-based Digital Contracts

Susanne Guth Gustaf Neumann Mark Strembeck
Department of Information Systems, New Media Lab
Vienna University of Economics and BA, Austria
{firstname.lastname}@wu-wien.ac.at

ABSTRACT

In this paper, we present our experiences concerning the enforcement of access rights extracted from ODRL-based digital contracts. We introduce the generalized *Contract Schema* (CoSa) which is an approach to provide a generic representation of contract information on top of rights expression languages. We give an overview of the design and implementation of the xoRELInterpreter software component. In particular, the xoRELInterpreter interprets digital contracts that are based on rights expression languages (e.g. ODRL or XrML) and builds a runtime CoSa object model. We describe how the xoRBAC access control component and the xoRELInterpreter component are used to enforce access rights that we extract from ODRL-based digital contracts. Thus, our approach describes how ODRL-based contracts can be used as a means to disseminate certain types of access control information in distributed systems.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability - Data mapping; D.4.6 [**Operating Systems**]: Security and Protection - Access controls; E.2 [**Data Storage Representation**]: Object representation; H.3.4 [**Information Storage and Retrieval**]: Systems and Software - Distributed systems; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection - Unauthorized access

General Terms

Security, Design, Experimentation

1. INTRODUCTION

A contract typically represents an agreement of two or more parties. The contract specifies rights and obligations of the involved stakeholders with respect to the subject matter of the respective contract. Contracts in the paper-world can be tailored to meet the needs of a specific business situation or to fit the requirements of individual contract partners. In principle, the same is true for digital contracts as they can be used in the area of digital rights management for example. Most often digital contracts are defined using special purpose *rights expression languages* (REL) as ODRL [16], XrML [8], or MPEG 21 REL [9] for instance.

In this connection one can differentiate between the "management of digital rights" and the "digital management of (arbitrary) rights". We especially focus on contracts that contain information on digital rights, i.e. rights which are intended to be controlled and enforced in an information system via a suitable access control service - in contrast to rights which are enforced by legislation or other "social protocols". In this paper, we consider digital contracts as a means to define and to disseminate certain types of access control information in distributed computing environments. In particular, we describe our experiences with the extraction and enforcement of access control information from ODRl-based digital contracts.

The remainder of this paper is structured as follows. In Section 2 we give an overview of the abstract structure of digital contracts. We especially describe how information within a digital contract is encapsulated in different contract objects. Section 3 then summarizes the contract processing procedures performed by a contract engine. Subsequently, Section 4 introduces the generalized contract schema CoSa and the software components we used to implement our system, before Section 5 shows how ODRL-based digital contracts are mapped to a runtime CoSa object model. Next, Section 6 describes the initialization of the xoRBAC access control service via a mediator component and the subsequent enforcement of the corresponding access rights. Section 7 gives an overview of related work, before we conclude the paper in Section 8.

2. STRUCTURE OF DIGITAL CONTRACTS

In this paper, we follow the terminology of [13] which suggests to model the content of digital contracts via interrelated *contract objects*, like Subject, Resource, Permission, Constraint, or Role (see Figure 1). Each contract object my contain attributes for further description. For example, subjects may have attributes such as name, address, or telephone number. Permission consists of an operation (e.g. play or print) and a resource (e.g. PDF-document or an MP3-file). Permissions are granted to subjects and can be narrowed by constraints. Constraints define invariants or side-conditions for access control decisions. In general, constraints may be applied to almost every part of an access control model. However, in our opinion defining constraints on permissions is esp. relevant for contract composition. Moreover, aside from the contract objects mentioned above, a digital contract itself carries own attributes such as a unique contract id, digital signatures, or comments for example.

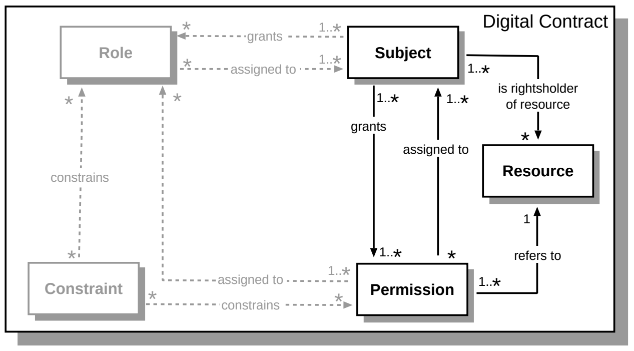


Figure 1: Required contract objects for the dissemination and enforcement of access control policies

Besides access control there is of course a number of other possible application areas for digital contracts, e.g. intellectual property rights (IPR) protection, accounting and sales statistics, or customer relationship management (CRM) (see e.g. [13]). However, since this paper focuses on access control we address contracts and contract objects only as far as they are needed for access control purposes. Subsequently, we describe the contract objects, depicted in Figure 1 in more detail (note that Figure 1 depicts optional contract objects in light grey):

- *Subject* is a mandatory contract object that appears at least once but typically twice in each contract. With respect to digital contracts, two different types of a subjects can be distinguished: A *rightsholder* is a subject who holds rights on the respective contract resource and may grant one or more rights to another contract party - the beneficiary. A *beneficiary* is the subject who receives rights from the rightsholder. In a digital contract a subject must always be identified through an unique identifier. In general, different types of values can be used to identify a contract party, e.g.:
  - A (globally but at least locally) unique identifier that identifies a certain *individual*, e.g. an X.500 distinguished name [17] or a Kerberos [38] established identity.
  - An unique identifier that identifies a certain *subject-type* (e.g. as defined in MARC 21 role code list [22]). A subject-type represents a number of individuals sharing one or more common characteristics. For example, a subject-type "faculty member" could represent each faculty member at the Vienna University of Economics and BA.

  Note that it is of course also possible to assign additional attributes to a subject such as name, or e-mail address for example.

- *Resource* is a mandatory contract object which appears at least once in a digital contract. A resource always has to be uniquely identified (globally, but at least locally), e.g. through a digital object identifier (DOI) [27].

- *Permission* is a mandatory contract object which appears at least once in digital contracts. A permission represents a $\langle operation, resource\rangle$ pair describing an operation that can be invoked on a specific object/resource. A simple example is a permission $\langle print, researchpaper\rangle$.

- *Role* is an optional contract object. With respect to role-based access control a role represents a set of permissions, i.e. permissions are assigned to roles and roles are assigned to subjects (parties) (see e.g. [11]). However, while roles are a convenient means to assign and manage permissions, permissions defined in digital contracts may also be directly assigned to a certain subject (without an intermediary role).

- *Constraint* is an optional contract object. A constraint can be defined as an invariant that must hold all the time (e.g. static separation of duties constraints), or as a side-condition which is evaluated dynamically and depends on certain runtime information, like time constraints for example.

- The *Digital Contract* object is the object which aggregates the above mentioned contract objects (see also Figure 1) and comprises (meta)information concerning the contract itself, for example:
  - *Unique Contract Identifier*: A mandatory attribute that contains a unique identifier for a specific contract, e.g. a URI [4]. If desired this attribute can be used to implement a duplication prevention mechanism and/or revocation lists for digital contracts.
  - *Digital signature*: A mandatory contract attribute. Each contract party has to sign a contract in order to conclude a contract. Thus, a digital contract contains at least one signature for each contract party (note that the beneficiary of a contract is not necessarily identical to the party signing the contract). For example, corresponding functions for XML-based contracts can be provided straightforwardly by applying standards like XML signature [3].
  - *Person Signature-certificate*: A mandatory attribute containing the signature certificate of a contract party, or a link to the respective certificate. Such a certificate allows to verify the digital signature of the corresponding signatories.
  - *Expiration date*: An optional attribute which determines the validity period of a contract through of an expiration date. If omitted the contract is assumed to be valid indefinitely, resp. until revoked or canceled by a dissolution contract.

3. CONTRACT PROCESSING

Figure 2 shows an activity diagram for the contract processing procedure. It esp. focuses on the extraction of access

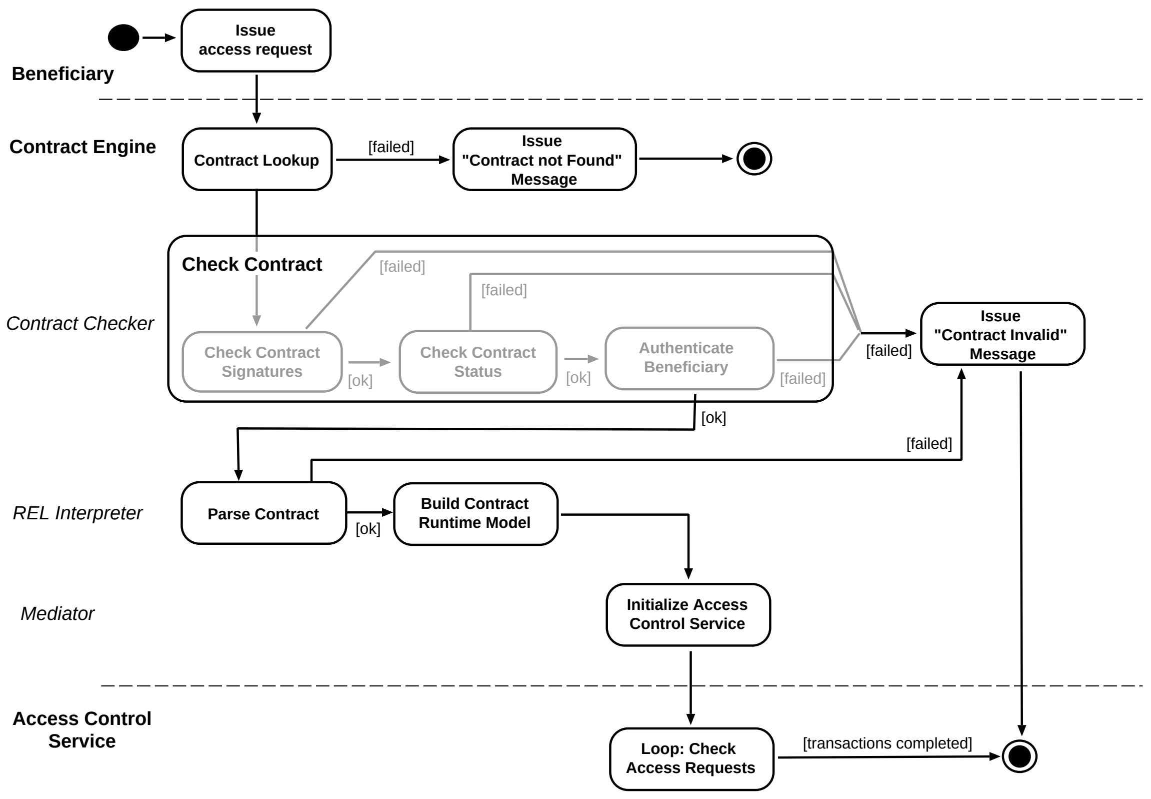


Figure 2: Basic activities in contract processing for access control

control information from digital contracts. Three different actors are involved in this process:

- The *beneficiary* (resp. a corresponding client program) who requests a service and presents a digital contract (see below).
- The *contract engine* which performs several checks on the contract and interprets the contract content (see e.g. [14]).
- The *access control service* which decides if the beneficiary may perform the requested operations according to the permissions granted through the presented contract.

The contract processing procedure is triggered by an access request. An access request expresses the demand to perform a specific operation/action on a particular object/resource. In our approach, each access request consists of a four valued vector $\langle subject, operation, object, contract-id\rangle$. Here, the contract-id is represented by an URI resp. an URL (see [4]). We bound the contract engine to an HTTP server which forwards corresponding requests to the contract engine (see Section 4); another possibility is, for example, to integrate a SOAP interface with the contract engine to allow for a direct processing of the above mentioned access requests. After an access request is received, the contract engine uses the respective contract-id to perform a *contract lookup* procedure to acquire the contract which is relevant for the current access request. The respective contract may either be already loaded, and is thus available from a local contract repository, or it is fetched from the URL indicated by the contract-id. After the contract is loaded into the contract engine the *contract checker* sub-component performs the following activities (cf. Figure 2):

- *Check Contract Signatures*: this activity verifies the digital signatures of the corresponding contract to ensure its integrity and authenticity.
- *Check Contract Status*: this activity inspects specific revocation lists in order to identify legally revoked (i.e. invalid) contracts and to prevent duplication, respectively "double spending" (see e.g. [23]), of digital contracts and the herein granted permissions. For example, the "double spending" prevention procedure is necessary if a contract defines a maximum number of uses for a certain digital good/service.
- *Authenticate Beneficiary*: since access control inevitably demands for a prior authentication of subjects, the contract engine performs an authentication procedure for the subject who issued a particular access request. Our approach does not demand for a particular authentication mechanism and can be applied with arbitrary authentication services, e.g. a Kerberos-based service, or an authentication infrastructure based on X.509-certificates.

If a digital contract successfully passed all checks, we consider the corresponding contract as *valid*. Note that Figure 2 shows the three sub-activities of "Check Contract" in light grey. We chose this coloration to indicate that these sub-activities and/or their succession are not necessarily the same for each contract engine. For example, the "Check Contract" activity could also include additional sub-activities (e.g. to check if the original grantor/rightsholder was actually allowed to assign the respective rights to the grantee/beneficiary), or one of the three sub-activities mentioned above could be performed by an other (external) software service. However, from our experiences the sequence shown in Figure 2 is a sensible set of activities that can and should be performed through a contract engine. If one of these basic activities fails, the contract engine issues a "Contract Invalid" message (see Figure 2) and all access requests relating to this particular contract are automatically denied without a need for further contract interpretation or additional access control measures. If, however, the check contract activity is completed successfully, the contract checker component forwards the contract to the REL Interpreter (rights expression language interpreter).

The REL interpreter parses the contract, extracts all (access control) relevant information, and builds a runtime contract object model. This object model may then be used to query the different contract objects and their respective attributes, e.g. unique ID of the beneficiary, his/her roles, the respective resources, the granted permissions to that resources, the constraints that apply to the granted permissions, etc. (the Sections 4.2 and 5 give a detailed description of the REL interpreter's functionality).

In the next step, a so-called Mediator component extracts the access control relevant contract information from the runtime model of the contract to initialize the corresponding access control service (see Figure 2). Finally, the access request of the beneficiary is forwarded to the access control service. The access control service performs the *Check Access Request* activity to evaluate the access request. In our implementation, we used the xoRBAC access control service which is briefly described in Section 4.1.

4. HIGH-LEVEL ARCHITECTURE AND SOFTWARE COMPONENTS

The high-level architecture depicted in Figure 3 shows the software components we used to implement our system: a *web server*, a *contract engine* (containing a contract checking module and a contract interpretation module), an *access control component*, and a *mediator* component. Note that, for the purposes of this paper, we assume that all software (and hardware) components used in our system are tamper resistant. Moreover, we assume that the different components authenticate each other and communicate via cryptographically secured channels. However, building real-world tamper resistant systems is of course a complex task - esp. in open, distributed environments (see also [2, 21])

The contract engine is intended to be a (web) server extension that enables the processing of XML-based digital contracts. Among other functionality the contract engine supports *contract checking* (cf. Section 3) and *contract interpretation*. The contract interpreter component basically uses a rights expression language (REL) interpreter to extract relevant information from digital contracts. In our system we utilize xoRBAC [24, 25] as access control service (see Section 4.1 and the xoRELInterpreter component as REL interpreter (see Section 4.2. The Mediator component interoperates with the contract engine, the web-server and the access control component (cf. Figure 3). It thus serves as a connector to glue the contract engine and external components (here xoRBAC) together. A concrete example of a mediator can be found in Section 6.

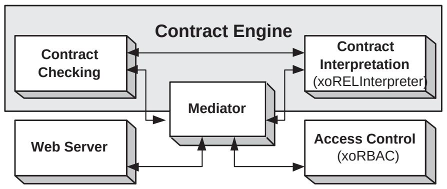


Figure 3: Abstract architecture of our system

The following sections describe the structure and functionality of xoRBAC and xoRELInterpreter in more detail. For the purposes of this paper, we esp. focus on the interaction between the contract interpreter component and the access control service. Therefore, we do not provide a detailed description of the contract checking component.

4.1 The xoRBAC Access Control Service

xoRBAC [24, 25] provides an RBAC service that can be used on Unix and Windows systems in applications providing C or Tcl linkage. xoRBAC is well suited to be used within a component framework. The xoRBAC component is implemented with XOTcl [26] which offers a dynamic programming environment for rapid application development. XOTcl itself is a Tcl [28] compliant component written in C. While originally developed as an RBAC service, xoRBAC was extended to provide a multi-policy access control system which can enforce RBAC, as well as DAC or MAC based policies including *conditional permissions*. Among other things xoRBAC provides the following features:

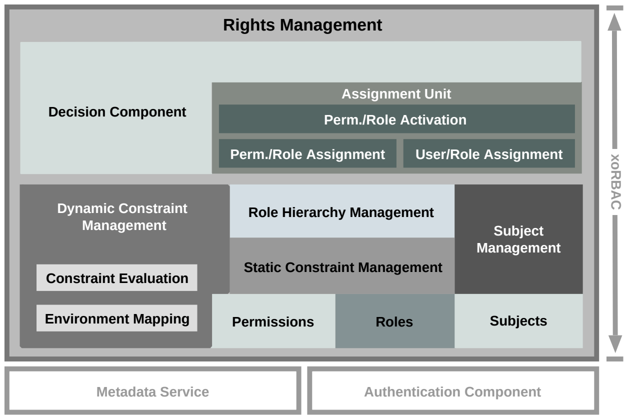


Figure 4: xoRBAC: conceptual structure

- Many-to-many role-to-subject, permission-to-role, and permission-to-subject assignment (and revocation).
- Definition of conditional permissions via context constraints.
- Arbitrary (DAG) role-hierarchies (permission-inheritance interpretation / inheritance hierarchies)
- Static separation of duties (SSD) constraints for both roles and permissions.

- Maximum and minimum cardinalities for both roles and permissions.

- Extensive review functions (introspection), e.g. subject-role review, permission-role review, subject-permission review.

Figure 4 depicts the conceptual structure of the xoRBAC component. With respect to this paper, the *dynamic constraint management* sub-system is of central significance. It comprises the *environment mapping* which captures context information via sensors, and the *constraint evaluation* which checks if the collected values match the context constraints associated with a certain conditional permission. Thereby it allows for the definition and enforcement of context constraints.

In essence, the *environment mapping* component comprises the *sensor library* of the xoRBAC access control service. It manages all sensors connected to xoRBAC. Therefore, every sensor must be registered in the sensor library before it can be used within xoRBAC. Each sensor provides one or more context functions. Thus, each context attribute that can be provided by a respective context function can be used to define xoRBAC context conditions. In this paper, we describe how xoRBAC can be used to enforce access control policies extracted from ODRL based digital contracts. A *context constraint* is defined through the terms context attribute, context function, and context condition:

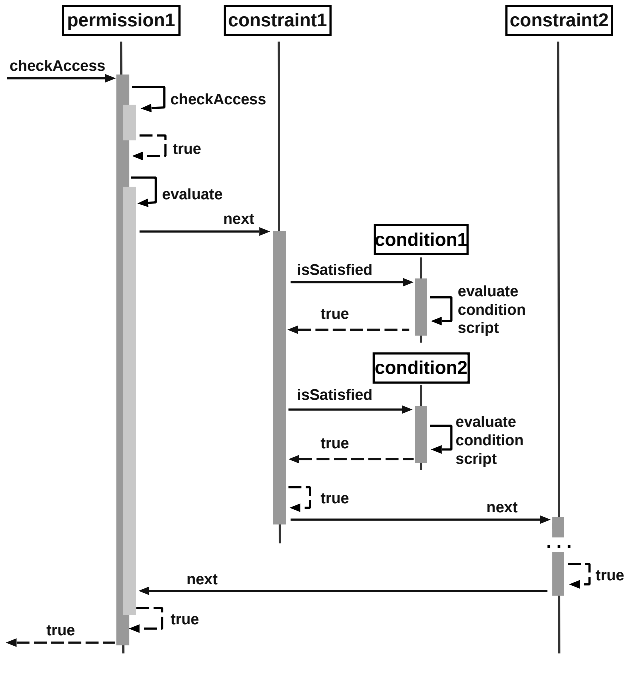


Figure 5: xoRBAC **access control decisions with context constraints**

- A *context attribute* represents a certain property of the environment whose actual value might change dynamically (like time, date, or session-data for example), or which varies for different instances of the same abstract entity (e.g. location, ownership, birthday, or nationality). Thus, context attributes are a means to make (exogenous) context information explicit. On the programming level each context attribute CA represents a variable that is associated with a $domain_{CA}$ which determines the type and range of values this attribute may take (e.g. date, real, integer, string).

- A *context function* is a mechanism to obtain the current value of a specific context attribute (i.e. to explicitly capture context information). For example, a function $date()$ could be defined to return the current date. Of course a context function can also receive one or more input parameters. For example, a function $age(subject)$ may take the subject name out of the $\langle subject, operation, object\rangle$ triple to acquire the age of the subject which initiated the current access request, e.g. the age can be read from some database.

- A *context condition* is a predicate (a Boolean function) that compares the current value of a context attribute either with a predefined constant, or another context attribute of the same domain. The corresponding comparison operator must be an operator that is defined for the respective domain. All variables must be ground before evaluation. Therefore, each context attribute is replaced with a constant value by using the according context function prior to the evaluation of the respective condition. Examples for context conditions can be $cond_1 : date() \leq$ "2003/01/01", $cond_2 : date() == birthday(subject)$, or $cond_3 : age(subject) > 21$.

- A *context constraint* is a clause containing one or more context conditions. It is satisfied iff all its context conditions hold. Otherwise it returns false.

With respect to the terms defined above, a *conditional permission* is a permission that is associated with one or more context constraints and grants access if and only if each corresponding context constraints evaluates to "true". Figure 5 shows a message sequence chart for access control decisions in xoRBAC including conditional permissions. For a detailed description of xoRBAC see [24, 25].

4.2 Rights Expression Language (REL) Interpreter

Rights Expression Languages (RELs) aim to provide a vocabulary and (partial) semantics for the expression of terms and conditions over (digital) assets. RELs aim to enable a machine-based processing of digital contracts (see e.g. [16]). In this paper, we describe how ODRL-based digital contracts can be used for the exchange of certain types of access control information in distributed systems. As already mentioned, the automated processing of contracts requires a REL interpretation service. In Section (*4.2.1*), we describe the *Generalized Contract Schema* (CoSa), before we introduce our implementation of an ORDL-based REL interpreter in Section (*4.2.2*).

4.2.1 Generalized Contract Schema (CoSa)

Digital contracts are often formulated using a rights expression languages that are specified via a corresponding XML Schema [10] (e.g. ODRL [16], XrML [8], or MPEG21-REL [9]). The document object model (DOM) [1] can be used to process XML documents. All elements of a static

XML (or HTML) document can be represented as an object tree, called DOM-tree (cf. Figure 7). For example, this tree contains objects like *Element, Entity*, or *Text*. For an XML-based digital contract the DOM-tree therefore contains all elements (represented by individual objects) that are written in a contract instance. The DOM-tree can be queried or modified via the DOM application programming interface (API) [1], or via XML-specific query language such as XPath [7] for example.

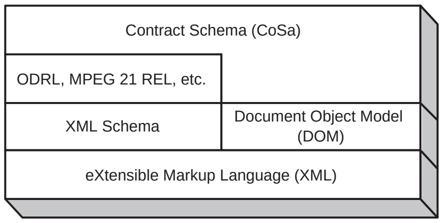


Figure 6: Layers of XML-based contract interpretation

To access and query digital contracts we introduce the Generalized Contract Schema (CoSa) that builds on top of the DOM- and the rights expression language layer (see Figure 6). It provides access to digital contracts (written in XML) via a generic runtime model that was built from the DOM-tree of the digital contract. The resulting runtime model of the Contract Schema is generic. This means that an actual CoSa model always relies on the same elements independent which rights expression language was used as input format. Thus, CoSa is an approach to build a generic representation of contract information on top of various rights expression languages. Therefore, ODRL or other rights languages can be used to define a linearized language-specific instance of CoSa.

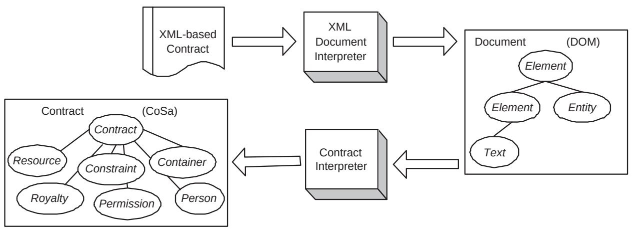


Figure 7: Building CoSa runtime models of digital contracts using DOM

CoSa represents all contract information as objects in a flat contract tree. A runtime contract tree is build by the means of dynamic object aggregation, which is a feature supported by XOTcl. The root objects (*Contract* or *Offer*) aggregate all related objects, such as *Party* (resp. Subject), *Resource*, *Permission*, *Constraint*, *Royalty*, and *Container* (cf. Figure 7). Relations between contract objects are represented by keys resp. IDs.

CoSa objects may have different kinds of attributes: *contract attributes, intrinsic attributes* and *application specific attributes.*

- *Contract attributes* store information from an XML contract instance. For example, *name, unique identifier, address* are typical contract attributes for the *Party* object, while *title, creator, unique identifier, format, size,* etc. are typical contract attributes for the *Resource* object. As the CoSa is independent from rights expression languages, attributes from a concrete REL-document instance have to be mapped to generalized CoSa attributes. For example, the ODRL vocabulary offers a constraint called *count* and XrML provides a term called *exerciseLimit* to narrow rights, e.g. if a right *play* may be executed only five times. Thus, a REL interpreter has to map language-specific attributes/elements to generic CoSa attributes/elements, e.g. to map ODRL count and XrML exerciseLimit attributes to the CoSa *limit* attribute. The catalog of CoSa contract attributes can be extended by additional vocabulary sets.

- *Intrinsic attributes* express relations between contract objects that result from the contract data model. In particular, the current CoSa implementation uses five different intrinsic attributes: *permissions, constraints, royalties, belongsto,* and *set*. For example, all permissions and royalties that have been assigned to a *Party* object can be found in the attribute *permissions* resp. *royalties* as a list of *Permission* resp. *Royalty* objects. Likewise, a *Permission* stores the identifiers of linked *Constraint* objects via a *constraints* attribute. The *set* attribute is used express a relation between multiple *Permission* or *Constraint* objects.

- *Application specific attributes.* Information that can not be represented via the first two attribute categories require an application specific extension of CoSa. Currently no additional application specific attributes are provided.

We describe an example of a CoSa runtime model in Section 5. The contract tree can be accessed via the CoSa API to query any information provided by a digital contract. The CoSa API offers a number of methods to retrieve any desired information for the processing software service (e.g. *getPermissions, getName, getUniqueID*). Example queries using methods offered by the CoSa API are given in Section 6.

When linearizing, parsing, and interpreting a certain contract, we must consider the language requirements of the respective rights language. For example, which mandatory elements exist in the respective language or what are the semantics of a certain element constellation. ODRL, for example, does not explicitly define mandatory elements. Moreover, depending on the rights to be expressed, ODRL provides multiple ways to phrase the same rights expression. Thus, an ODRL interpreter has to be aware of the different variants.

Extracted contract information may be queried and further processed in other software services. In the case study presented in this paper we extract information for further processing in the xoRBAC access control service (see Section 4.1). For the time being the CoSa API simply allows to query but not to modify contracts. The modification of contracts is a sensitive issue and brings up new challenges for the corresponding software components, e.g. with respect to access and modification rights to a digital contract itself or concerning the validity of the digital contract.

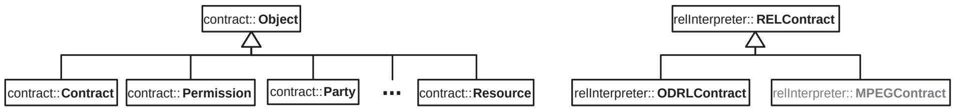


Figure 8: Packages and classes of the REL Interpreter

4.2.2 REL Interpreter Implementation

The xoRELInterpreter is intended to be a general purpose contract interpreter and provides the transformation of XML-based digital contracts into instances of the generalized contract schema (CoSa). Like xoRBAC, the xoRELInterpreter is implemented with XOTcl and can be used on Unix and Windows systems within applications providing C or Tcl linkage.

To interpret XML documents the xoRELInterpreter makes use of tDOM [18] library which provides a sophisticated implementation of the document object model (DOM), XPath, and other W3C standards. The xoRELInterpreter consists of two packages the *relInterpreter-*, and the *contract* package (see Figure 8). The abstract class RELContract (part of the relInterpreter package) provides the external CoSa API. Thus, every subclass of RELContract (e.g. the class ODRLContract) has to implement the abstract CoSa interface defined in RELContract. We implemented the ODRLContract class to provide an ODRL Interpreter conforming to [16]. The *contract* package provides class definitions for contract objects (*Party* (resp. Subject),*Resource, Permission*, etc., see Section 2) defined by the generalized contract schema. We chose ODRL for our implementation because ODRL relies on an open and freely available specification. Furthermore, ODRL offers a straightforward approach for the expression of rights and is well-known in the research community. Moreover, ODRL is the favored rights expression language in the educational domain.

5. GENERALIZED CONTRACT SCHEMA OF REL-BASED DIGITAL CONTRACTS

This section illustrates the interpretation of digital contracts defined through a rights expression language (REL) and their transformation into the generalized contract schema (CoSa). To demonstrate the functionality of CoSa we first give an example of an ODRL-based contract and its transformation into CoSa software objects. Afterwards, we briefly show a sample XrML representation of the same CoSa software objects.

5.1 Generalized Contract Schema of an ODRL-based Digital Contract

Figure 9 depicts an example of an ODRL contract. ODRL is defined via two XML schemas (see also[16]) - the expression language schema (prefix: o-ex), and the data dictionary schema (prefix:o-dd).

Our ODRL contract contains a *license* sometimes also called *digital ticket* (see e.g. [12, 13, 35]). The license depicted in Figure 9 grants the rights to `display` and `print` the asset `rossi-12345` to the party `Mary Smith` with the (locally) unique id `msmith`. The license also carries some context information: the *physical location*, the *license id* and a *remark*. The `display` and `print` rights are narrowed by the following constraints: the permission `print` may only be executed *two times* and the right `display` may only be performed on a specific CPU - *Intel-12345*. Moreover, in order to actually obtain both rights the consumer first has to pay the amount of *AUD 20.00* and a *10 % GST tax*.

The contract above results in a structure consisting of several CoSa contract objects (cf. Figure 10), aggregated to an instance of the *Contract* type that carries the context information of the license. The *Party*, *Resource*, *Constraint*, and the *Permission* objects are aggregated within the root object *Contract*. The relations among the different runtime contract objects are expressed via keys/IDs, stored via intrinsic attributes (cf. Section 4.2.1). For example, the party *person-001* "owns" the *Permission* objects *p-001* and *p-002* (see Figure 10). The *Permission* object *p-001* is narrowed by the constraints *c-001, c-003*, and *c-004*, whereas the *Permission* object *p-002* is narrowed by the constraints *c-002, c-003*, and *c-004*. The *Condition* object *c-002* "belongs to" the *Permission* object *p-002*, and the *Condition* object *c-003* is related to the *Permission* objects *p-001* and *p-002*.

```
<?xml version="1.0" encoding="UTF-8"?>
<o-ex:rights xmlns:o-dd="http://odrl.net/1.1/ODRL-DD"
   xmlns:o-ex="http://odrl.net/1.1/ODRL-EX"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://odrl.net/1.1/ODRL-DD
                            H:\Daten\odrl\ODRL-DD-11.xsd">
  <o-ex:agreement>
    <o-ex:context>
      <o-dd:uid> license-12345 </o-dd:uid>
      <o-dd:pLocation> Sydney, Australia </o-dd:pLocation>
      <o-dd:remark> Transacted by Example.Com </o-dd:remark>
    </o-ex:context>
    <o-ex:asset>
      <o-ex:context>
        <o-dd:uid> rossi-12345 </o-dd:uid>
      </o-ex:context>
    </o-ex:asset>
    <o-ex:permission>
      <o-dd:display>
        <o-ex:constraint>
          <o-dd:cpu>
            <o-ex:context>
              <o-dd:uid> Intel-12345 </o-dd:uid>
            </o-ex:context>
          </o-dd:cpu>
        </o-ex:constraint>
      </o-dd:display>
      <o-dd:print>
        <o-ex:constraint>
          <o-dd:count> 2 </o-dd:count>
        </o-ex:constraint>
      </o-dd:print>
      <o-ex:requirement>
        <o-dd:prepay>
          <o-dd:payment>
            <o-dd:amount o-dd:currency="AUD"> 20.00</o-dd:amount>
            <o-dd:taxpercent o-dd:code="GST"> 10.00</o-dd:taxpercent>
          </o-dd:payment>
        </o-dd:prepay>
      </o-ex:requirement>
    </o-ex:permission>
    <o-ex:party>
      <o-ex:context>
        <o-dd:uid> msmith </o-dd:uid>
        <o-dd:name> Mary Smith </o-dd:name>
      </o-ex:context>
    </o-ex:party>
  </o-ex:agreement>
</o-ex:rights>
```

Figure 9: Example of an ODRL License

For the interpretation of ODRL instances, each permission tag is assigned a set id. Aggregated permission elements, such as play or display, thus get the same set id (cf. Figure 9. Through the set id we can distinguish if a con-

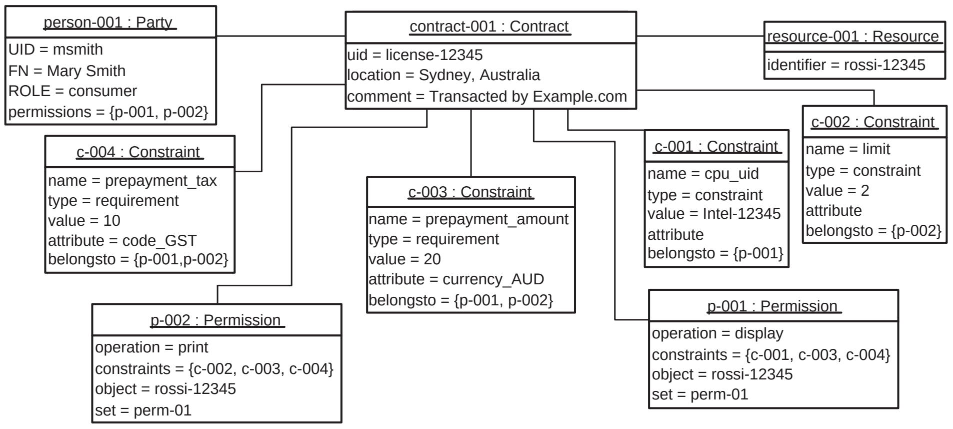


Figure 10: Contract Objects: Properties of a RELContract

straint applies to a single permission element or to a set of permission elements. For example, the advance payment of AUD 20 and 10 percent tax applies only once to the permission set *perm-01* (see Figure 10). Note that ODRL-specific attributes have been mapped to the generic CoSa attributes. For example, the ODRL attribute "permission" is mapped to the generic CoSa attribute "operation". In turn, in CoSa a permission is the combination of on operation and an object.

```
<?xml version="1.0" encoding="UTF-8"?>
<license xmlns="http://www.xrml.org/schema/2001/11/xrml2core"
    xmlns:sx="http://www.xrml.org/schema/2001/11/xrml2sx"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cx="http://www.xrml.org/schema/2001/11/xrml2cx"
    xmlns:ex="http://www.xy.org/ex-schema/2003/xrml-ex"
    xsi:schemaLocation="http://www.xrml.org/schema/2001/11/xrml2cx
              ..\schemas\xrml2cx.xsd">

  <title> This license is transacted by Exampple.Com </title>
  <ex:uniqueID type="Ariadne">  license-12345 </ex:uniqueID>
  <ex:physicalLocation>Sydney, Australia</ex:physicalLocation>

  <inventory>
    <ex:consumer licensePartId = "msmith">
      <ex:uniqueID type="Uni-ID">  msmith </ex:uniqueID>
      <ex:name>  Mary Smith </ex:name>
    </ex:consumer>
    <ex:electronicBook licensePartId = "rossi-12345">
      <ex:uniqueID type="Res-ID">  rossi-12345 </ex:uniqueID>
    </ex:electronicBook>
  </inventory>

  <grant>
    <grantGroup>

      <ex:consumer licensePartIdRef="msmith"/>

      <grant>
        <ex:electronicBook licensePartIdRef="rossi-12345"/>
        <cx:print/>
        <ex:device>
           <ex:uniqueID type="cpu">  Intel-12345 </ex:uniqueID>
        </ex:device>
      </grant>

      <grant>
        <ex:electronicBook licensePartIdRef="rossi-12345"/>
        <ex:display/>
        <ex:limit>  2 </ex:limit>
      </grant>
    </grantGroup>

    <sx:fee>
      <sx:cash currency=AUD>  20.00 </sx:cash>
      <ex:tax code=GST>  10.00 </ex:tax>
    </sx:fee>

  </grant>
</license>
```

Figure 11: XrML linearization of CoSa objects

Moreover, ODRL distinguishes three different "constructs" to narrow *Permissions*: requirements, constraints, and conditions (cf. [16]). These constructs, however, are not supported by other rights expression languages, e.g. XrML [8]. Therefore, in CoSA, ODRL requirements, ODRL constraints, and ODRL conditions are mapped to a generic *Constraint* type.

To interpret the ODRL contract shown in Figure 9, and to build a corresponding CoSa instance we use the following simple command line (when instantiating the ODRLContract object the digital contract, or its location is given as a parameter): `ODRLContract c1 ../Ebook2.xml`

Subsequently, the CoSa API can be used to query the object `c1`, e.g. to extract contract parties and their permission. An example using the CoSa API is given in the next section.

5.2 Example for a Linearization of a Contract Schema into XrML

The interpretation of an ODRL document results in a set of generic CoSa objects. In other words, a contract written in a different rights expression language (e.g. XrML) containing the same contract information would result in the same CoSa contract objects and attributes. For example, Figure 11 shows an XrML document containing the same contract information as the ODRL document presented in the previous section.

Therefore, CoSa is, on the one hand, an approach to build a generic representation of contract information. On the other hand, it can be applied to convert contract information from one rights expression language to another. However, note that the current version of the CoSa implementation does not (yet) support the "reverse mapping" of CoSa objects to ODRL or XrML contracts.

6. ACCESS CONTROL DECISIONS BASED ON ODRL CONTRACTS

We distinguish two basic forms how digital contracts may be applied for access control purposes. For the purposes of this paper we esp. investigated the first alternative. However, we also discuss some issues related to the second variant:

1. The access control service permanently *imports* all access control information from the digital contract - "it instantiates the contract". For example, this is sensible in a system where users frequently request access

operations and where relatively small time intervals elapse between two requests of the same user.

2. The access control service uses digital contracts as an additional *external control set* and *queries* the contract on demand each time a contract-dependent access is requested. This is sensible, for instance, in systems where the same user seldom issues access requests, like in a video on demand system for example.

Figure 12 indicates how information from ODRL-based digital contracts is mapped to xoRBAC objects. The mappings of `party` to *Subject*, and `asset` to *Object* are straightforward, while a ODRL `permission` is equivalent to an *Operation* in xoRBAC. However, in this paper we use the term *Permission*, as widely accepted, to indicate an $\langle operation, object \rangle$ pair (see e.g. [11]). As mentioned in Section 5, ODRL `requirements` and `constraints` are mapped to CoSa *Constraint* objects. In xoRBAC each of these three types (requirement, condition, constraint) is mapped to context constraints (cf. Section 4.1). In ODRL, roles are most often not explicitly defined but result from the interpretation of the contract. For example, a party, that has no rightsholder element, is (automatically) assigned to the local contract role "beneficiary". Therefore, Figure shows no direct mapping of an ODRL element to an xoRBAC *Role* object.

```
requestAccess (subject, operation, object, contract-id) {
  #contract lookup
  set contractInstance [contractLookup(contract-id)]
  #check contract
  set contractOK [checkContract(contract-id]
  #interpret contract and initialize xoRBAC instance "rm"
  ODRLContract c1 $contractInstance
  RightsManager rm
  if {$contractOK == true} {
     set contracts [c1 getContracts]
     foreach c $contracts {
        set assets [c1 getAssets $c]
        set consumers [c1 getConsumers $c]
        foreach asset $assets {
           set assetId [c1 getUniqueID $asset]
           foreach con $consumers {
              set conID [c1 getUniqueID $con]
              rm createSubject $conID
              set perms [c1 getPermission $con]
              foreach p $perms {
                 set perm [c1 getName $p]
                 rm createPermission $perm
                 rm subjectPermAssign $conID $perm
              }
           }
        }
     }
  } else {
    #contract checks not successfull, access denied.
    return false
  }
  #checkAccess - performed by access control service
  set success [rm checkAccess $subject $operation $object]
  if {$success} {
    return true  #access granted
  } else {
    return false #access denied
  }
}
```

Figure 13: Excerpt of a mediator source code

To actually perform the mapping procedure explained above, xoRBAC and the xoRELInterpreter must be linked together on a technical level. Figure 13 shows a simple code example of a special purpose mediator (written in the XOTcl programming language [26]) which maps ODRL contract information to an instance of the xoRBAC service (represented by the `RightsManager` instance `rm` - see Figure 13). In this example the following CoSa API methodes are used:

- `getContracts()`: Returns a list of all *Contract* objects registered for the current ODRL instance (an ODRL interpreter instance can contain more than one contract).
- `getAssets(contract)`: Returns a list of all *Asset* objects included in a special contract.
- `getConsumers(contract)`: Return a list of all *Consumer* objects included in a special contract.
- `getUniqueID(object)`: Return the value of the *id* attribute of the respective CoSa `object`. The `object` could be of any valid CoSa type (e.g. Party, Resource, or Permission).
- `getPermission(consumer)`: Returns a list of all *Permission* objects, assigned to the respective `consumer`.
- `getName(object)`: Returns the value of the *name* attribute of the respective CoSa `object`.

The simplified example mediator shown in Figure 13 controls the processing of a complete access request, including contract lookup, contract checking, contract interpretation, and processing of the actual access request via xoRBAC - as mentioned above a mediator is used to glue together to other components (see Section 4). In particular, the CoSa API provides methods to obtain the relevant access control information from a CoSa runtime model, e.g. of an ODRL-based digital contract (see Section 4.2), and the mediator initializes the corresponding xoRBAC instance accordingly (method calls: `createSubject`, `createPermission`, `subjectPermAssign` in Figure 13). However, for the sake of simplicity the example above omits the source code for the creation and assignment of *Context Constraints* (see [25]).

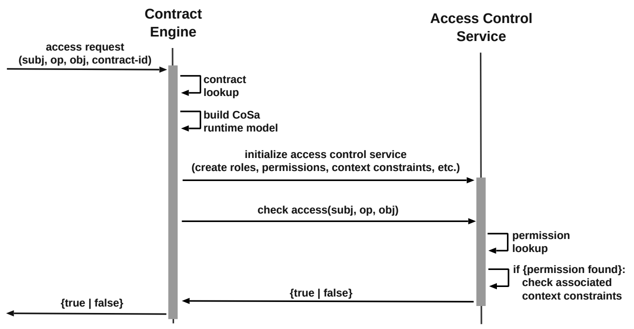
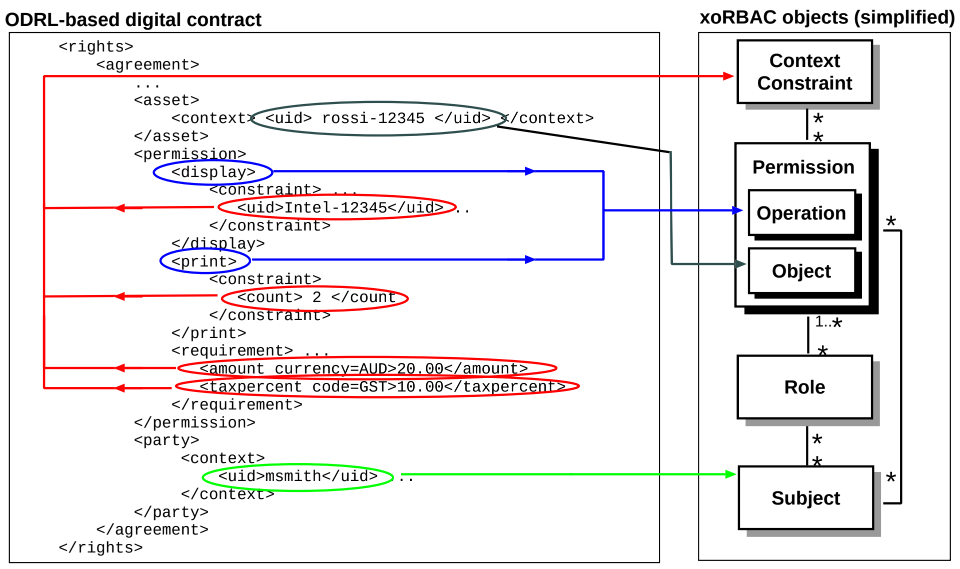

Contract Engine
Access Control Service
access request (subj, op, obj, contract-id)
contract lookup
build CoSa runtime model
initialize access control service (create roles, permissions, context constraints, etc.)
check access(subj, op, obj)
permission lookup
if {permission found}: check associated context constraints
{true | false}
{true | false}

Figure 14: Access control decisions based on digital contracts - simplified message sequence chart

Subsequent to the set-up phase, xoRBAC is ready to decide about access requests via the `checkAccess` method. Figure 14 depicts the corresponding flow of events as a message sequence chart. An xoRBAC access request consists of a $\langle subject, operation, object \rangle$ triple. First, xoRBAC checks if the corresponding subject owns a permission, which grants the respective access request. If so, it checks the context constraints associated with this particular permission. The evaluation of xoRBAC context constraints is performed by the constraint evaluation sub-component which, in turn,



```
<rights>
    <agreement>
        ...
        <asset>
            <context> <uid> rossi-12345 </uid> </context>
        </asset>
        <permission>
          <display>
              <constraint> ...
                <uid>Intel-12345</uid> ..
              </constraint>
          </display>
          <print>
              <constraint>
                <count> 2 </count
              </constraint>
          </print>
          <requirement> ...
             <amount currency=AUD>20.00</amount>
             <taxpercent code=GST>10.00</taxpercent>
          </requirement>
        </permission>
        <party>
             <context>
                 <uid>msmith</uid> ..
             </context>
        </party>
    </agreement>
</rights>
```

Figure 12: Mapping of ODRL contract information to xoRBAC **objects**

uses the environment mapping sub-component to capture context data, as time, date, or IP-address for example (see Section 4.1). If all context-constraints linked to a permission evaluate to **true** all prerequisites are fulfilled and the decision component returns **true** to indicate that the corresponding access request is allowed (see Figure 5).

In other words: it is not sufficient for a subject to own a specific permission in order to grant a corresponding access request. Additionally, the contract from which the permission was extracted must be valid and all context constraints associated with the granting permission must evaluate to **true**. Therefore, contract-dependent access control has at least two specific characteristics:

- A dynamic and per access status/validation check of the digital contract a specific permission originates from. Here, the contract can be used as a means to assign (or revoke) rights to (from) users, to disseminate access control information in a distributed computer network.
- A dynamic and per access environment mapping - because context constraints are based on context parameters whose actual values change relatively often, or which vary for different individuals or for different instances of the same abstract entity (see also Section 4.1).

7. RELATED WORK

The eXtensible rights markup language (XrML) [8] is a rights expression language developed by ContentGuard [1] which, in turn, is a joint venture of Xerox and Microsoft. XrML Version 2.0 was selected by the moving pictures expert group (MPEG) as the basis for development of the MPEG 21 Part 5 [2] standard. ContentGuard has released a XrML Software Development Kit (SDK) that includes an example license interpreter and an example condition validator.

- The "license interpreter" basically provides a single method: *validateGoal(PrincipalList, RightsList, ResourceList)*. This method is fetches all conditions from an XrML document that are linked to the triple $\langle principal, resource, right\rangle$. For example, if the respective XrML license contains the triple $\langle Mary, hit.mp3, play\rangle$ the method fetches all conditions associated with this triple (e.g. a restriction to play the music file for five times only). All conditions extracted via the validateGoal method are further processed by the "condition validator".
- The "condition validator" checks if the corresponding conditions are met, for the example mentioned above, it decides whether Mary may play the hit.mp3 or not. For the time being, the XrML condition validator is capable of checking two constraints: time interval and exercise limit.

The XrML "license interpreter" and the "condition validator" are closely coupled and are not designed to operate separately. The interpreter does not provide additional functions to make contract information available to other applications, like access control services for example. Thus, the current version of the XrML SDK is a proof-of-concept implementation that is focused on one particular application of the rights expression language XrML. In contrast to that, we developed a general approach for the interpretation of rights expression languages. In particular, the generalized contract schema CoSa can be applied to extend arbitrary applications (providing C or Tcl linkage) with contract processing abilities - in specific different components/applications are glued together via a customized mediator component (see Section 4). Thereby, the CoSa itself is independent from certain rights expression languages and from the applications that use/process contract data.

[1]http://www.contentguard.com
[2]The ISO/IEC working group in charge of the development of standards for coded representation of digital audio and video, http://www.chiariglione.org/mpeg/)

Shand and Bacon [37] present a contract framework that includes an abstract contract protocol for contract exchange and an accounting language (based on the Python scripting language) for the specification of accounting policies. Contracts define the resources that are exchanged between contracting parties, e.g. CPU time, network bandwidth, or money. Contracts must be signed by all contract parties to be valid. A peculiarity of their approach is that trust is treated as a special type of resource which influences the conclusion of a contract. The trustworthiness of a certain party is continuously adapted according to her/his contractual fidelity.

Park and Sandhu propose a high-level model for the definition of usage control policies [30]. Usage control (UCON) works on the principle that digital objects are encapsulated in a secure "digital container". Information within such a digital container can only be accessed through specific (tamper-resistant) soft- and/or hardware devices by feeding in a set of access rights approved by the originator of the corresponding container. The set of access rights can be seen as a license or a contract between the originator and the recipient/consumer. Their model is, however, defined on a high level of abstraction and must be refined before it may serve as a basis for the definition of actual UCON policies. In [29], Park and Sandhu describe an approach to combine usage control and originator control. Originator control was already mentioned in [20] and is a concept which requires that recipients obtain the originator's approval prior to the re-dissemination of digital objects. In their approach, "licenses" are digitally signed certificates defining the usage rights for digital objects. Users can access digital objects only according to their license. Tickets are used to transfer "re-dissemination" rights for digital objects.

In the work of Keller et al. [19] a management architecture for specifying, deploying, monitoring, and enforcing service contracts is proposed to provide a basis for service level agreements. Contracts are concluded between service providers and a service integrator, and contracts contain agreements about quality of service (QoS) attributes. Keller at al. define object classes that represent the contract model. Their contract model is tailored to the needs of service level agreements, and thus contains different contract objects as the contract model discussed in this paper (see Section *4.2.1*). However, their model also contains basic contract objects that can be found in CoSa, such as provider, customer, and service, as well as objects that represent the guaranteed service parameters (rights). Keller et al. do not envision the exchange of contract information between the involved components in a standardized format, such as a rights expression language.

In [5], Beugnard et al. introduce a general model of software contracts that aims at increasing trust and reliability between software components. To conclude contracts between components, every component publishes a feature set to describes its services in a common language (e.g. CORBA IDL). Contracts are established between a client and server component in a negotiation phase where the contract parties agree on certain services. The work provides a basic interface description for the negotiation phase. Beugnard et al. suggest an "XML-formatted description of the contracts" that is applied for negotiation purposes. This is similar to the approach presented in this paper, and a rights expression language in combination with a REL interpreter could also be used to express the service level agreement information and subsequently process it in the negotiation and execution process.

Sandhu and Park introduce so called smart certificates for attribute services on the Web [31]. They use the extension field of X.509v3 certificates to bind attributes to a subject (party). In an other contribution Sandhu and Park [32] present an implementation where the extension field of a X.509v3 certificate is used to assign role information to a subject. Based on the subject's role information, web servers use roles instead of a user's identity for access control purposes. In [33] Park and Sandhu introduce secure cookies. They use cryptographic techniques to enrich cookies (resp. information stored via cookies) with integrity, authentication, and confidentiality. Web servers can use secure cookies to store sensitive data on a client computer and reuse the corresponding information in future interactions with the same client.

XACML [36] describes a policy language and an access control decision request/response language (both written in XML). The policy language aims at defining authorization policies, i.e. rules defining what operations a specific subject is allowed to perform on a certain set of objects. Thus, XACML aims to support the definition of policies that can be used (interpreted) by different access control mechanisms, e.g. mechanism providing role-based access control. The main focus of rights expression languages is different from XACML in that digital contracts often include supplemental information like "notes" on certain terms and conditions that are intended to be interpreted by human users. However, it is likely that most information expressed through a rights expression language can be mapped to XACML.

8. CONCLUSION AND FUTURE WORK

In this paper, we presented our experiences with the enforcement of access rights which are extracted from ODRL-based digital contracts. We presented the generalized contract schema CoSa which provides a means for the generic representation of digital contracts formulated in arbitrary rights expression languages. Moreover, we introduced the xoRELInterpreter software component which (in its current version) is able to interpret digital contracts based on ODRL and to build a runtime CoSa object model from arbitrary ODRL documents. We use so called mediator components to query a runtime CoSa model and to interact with other software components. A mediator therefore glues our contract engine and external components together to form contract-aware applications. Among other things, we described our experiences in the design and implementation of a mediator between the xoRBAC access control component and the xoRELInterpreter component. In specific, we have shown how these components are used to extract access control information from ODRL documents and to actually enforce the corresponding policies. Our approach uses ODRL-based contracts as a means to disseminate access control information in distributed systems.

We currently extend the xoRELInterpreter component to support XrML and MPEG REL based digital contracts. Furthermore, we further investigate the need for standardized vocabulary in the area of rights expression languages and continue to improve the generalized contract schema (CoSa) in order to provide a generic framework for the description and interpretation of digital contracts.

An other problem area we plan to address in our future work is the need for rights control licenses. A *rights control license* certifies the control permissions of a particular subject. A control permission is an administrative permission which enables its owner to grant a certain access right to other subjects. This means that a contract party X must only grant a specific access permission to an other contract party Y if X is in possession of the corresponding control permission. As long as contracts are only used within a closed system where the resp. system authority accepts "self-signed" contracts only, there is no urgent need for the use of rights control licenses. However, in a distributed system with several different local authorities, rights control licenses are needed in order to guarantee that each contract party does only assign (or revoke) rights to (from) an other contract party if he is authorized to do so (i.e. only if the corresponding authority owns a respective control permission). Nevertheless a number of problems may arise in case rights control licenses are used, e.g. concerning the delegation and or (re)selling of contract rights.

Moreover, similar to contracts in the paper-world, digital contracts can become invalid for some reason. Thus, a means (and infrastructure) for distributed contract and/or rights revocation is needed. This, again, is similar to certificate infrastructures for public-key cryptography, where so called revocation-lists are used to publish revoked or invalid certificates (see e.g. [15]). Eelated problems arise from the fact that a digital contract must be digitally signed in order to assure its integrity and authenticity. Nevertheless, public-key certificates may expire or become invalid for various reasons. Therefore, it is not trivial to define what happens to digital contracts that were signed with a certain public-key if the corresponding certificate becomes invalid. An other problem is contract duplication, respectively double spending of the rights granted through a specific contract, which is quite similar to the double-spending problem of electronic money (see e.g. [6, 34]).

9. REFERENCES

[1] A. Le Hors, et al. Document Object Model (DOM) Level 2 Core Specification. http://www.w3.org/DOM/, November 2000.

[2] R. Anderson and M. Kuhn. Tamper Resistance - a Cautionary Note. In *Proc. of the 2nd USENIX Workshop on Electronic Commerce*, November 1996.

[3] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon. XML-Signature Syntax and Processing. http://www.w3.org/TR/xmldsig-core/, February 2002. W3 Consortium Recommendation.

[4] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. IETF RFC 2396, Standards Track, http://www.ietf.org/rfc/rfc2396.txt, August 1998.

[5] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer Magazine*, 32(7), July 1999.

[6] J. Camenisch, J.M. Piveteau, and M. Stadler. An Efficient Fair Payment System . In *Proc. of ACM Conference on Computer and Communications Security*, 1996.

[7] J. Clark and S. DeRose. XML Path Language (XPath). http://www.w3.org/TR/xpath, November 1999. W3 Consortium Recommendation.

[8] ContentGuard Inc. eXtensible rights Markup Language (XrML), Version 2.0. http://www.xrml.org/, November 2001.

[9] T. DeMartini, X. Wang, and B. Wragg. MPEG-21 Working Documents - Part 5 & Part 6, MPEG-21 Rights Expression Language. http://www.chiariglione.org/mpeg/working_documents.htm, March 2003.

[10] D. C. Fallside. eXtenisble Markup Language (XML) Schema Specification 1.0. http://www.w3.org/TR/XML/Schema/, May 2001. W3 Consortium Candidate Recommendation.

[11] D.F. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security*, 4(3), August 2001.

[12] K. Fujimura and Y. Nakajima. General-Purpose Digital Ticket Framework. In *Proc. of the 3rd USENIX Workshop on Electronic Commerce*, September 1998.

[13] S. Guth, G. Neumann, and M. Strembeck. Toward a Conceptual Framework for Digital Contract Composition and Fulfillment. In *Proc. of the International Workshop for Technology, Economy, Social and Legal Aspects of Virtual Goods*, May 2003.

[14] S. Guth, B. Simon, and U. Zdun. A Contract and Rights Management Framework Design for Interacting Brokers. In *Proc. of the 36th Hawaii International Conference on System Sciences*, January 2003.

[15] R. Housley and T. Polk. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*. John Wiley & Sons, 2001.

[16] R. Iannella. Open Digital Rights Language (ODRL), Version 1.1. http://odrl.net, August 2002.

[17] ITU-T. ITU-T Recommendation X.500: Information Technology-Open Systems Interconnection-The Directory: Overview of Concepts, Models and Services, 1993.

[18] J. Loewer and R. Ade. tDOM (DOM Implementation). available at: http://www.tdom.org/, 2003.

[19] A. Keller, G. Kar, H. Ludwig, A. Dan, and J.-L. Hellerstein. Managing Dynamic Services: A Contract Based Approach to a Conceptual Architecture. In *Proc. of the 8th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2002.

[20] C.E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3), September 1981.

[21] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[22] Machine Readable Cataloging (MARC) Standards Office. MARC Code List for Relators, Sources, Description Conventions. http://www.loc.gov/marc/relators/, January 2003.

[23] G. Medvinsky and C. Neuman. NetCash: A Design for Practical Electronic Currency on the Internet. November 1993.

[24] G. Neumann and M. Strembeck. Design and

Implementation of a Flexible RBAC-Service in an Object-Oriented Scripting Language. In *Proc. of the 8th ACM Conference on Computer and Communications Security (CCS)*, November 2001.

[25] G. Neumann and M. Strembeck. An Approach to Engineer and Enforce Context Constraints in an RBAC Environment. In *Proc. of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2003.

[26] G. Neumann and U. Zdun. XOTcl, an Object-Oriented Scripting Language. In *Proc. of Tcl2k: 7th USENIX Tcl/Tk Conference*, February 2000.

[27] National Information Standards Organization (NISO). Syntax for the Digital Object Identifier. http://www.niso.org/standards/, December 2000.

[28] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[29] J. Park and R. Sandhu. Originator Control in Usage Control. In *Proc. of the 3rd International Workshop on Policies for Distributed Systems and Networks*, June 2002.

[30] J. Park and R. Sandhu. Towards Usage Control Models: Beyond Traditional Access Control. In *Proc. of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, June 2002.

[31] J.S. Park and R.Sandhu. Smart Certificates: Extending X.509 for Secure Attribute Services on the Web. In *Proc. of 22nd National Information Systems Security Conference (NISSC)*, October 1999.

[32] J.S. Park and R. Sandhu. RBAC on the Web by Smart Certificates. In *Proc. of the ACM Workshop on Role-Based Access Control*, October 1999.

[33] J.S. Park and R. Sandhu. Secure Cookies on the Web. *IEEE Internet Computing*, 4(4), July/August 2000.

[34] B. Pfitzmann and M. Waidner. Strong Loss Tolerance of Electronic Coin Systems. *ACM Transactions on Computer Systems*, 15(2), May 1997.

[35] L.R. Rivest. Electronic Lottery Tickets as Micropayments. In R. Hirschfeld, editor, *Financial Cryptography*. Springer Verlag, November 1997.

[36] S.Godik and T.Moses (eds.). eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS Standard, February 2003.

[37] B. Shand and J. Bacon. Policies in Accountable Contracts. In *Proc. of the 3rd International Workshop on Policies for Distributed Systems and Networks*, June 2002.

[38] J.G. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proc. of the USENIX Winter Conference*, February 1988.