Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report No. 517

Controlling Access to Documents: A Formal Access Control Model

Paul E. Sevinç and David Basin
Department of Computer Science

May 4, 2006

This work was partially supported by the Zurich Information Security Center. It represents the views of the authors.

Contents

1 **Introduction** 3
- 1.1 Contributions . . . 3
- 1.2 Organization . . . 4

2 **Requirements** 4

3 **Document Content Model** 5
- 3.1 Content Model . . . 5
  - 3.1.1 Data Types and Auxiliary Functions . . . 6
  - 3.1.2 Containers (State) and Initialization Operations . . . 7
  - 3.1.3 Operations . . . 7
- 3.2 Example Refinements . . . 11
  - 3.2.1 File System . . . 11
  - 3.2.2 LDAP Directory . . . 12

4 **Document Policy Model** 12
- 4.1 Informal Description . . . 13
  - 4.1.1 Policy Language . . . 13
  - 4.1.2 Access-Control Architecture . . . 14
- 4.2 Formal Specification . . . 15
  - 4.2.1 Data Types . . . 15
  - 4.2.2 Containers (State) and Initialization Operations . . . 16
  - 4.2.3 Operations on Nodes and Auxiliary Functions . . . 17
  - 4.2.4 Operations on Attributes . . . 21
  - 4.2.5 Policy-specific Operations . . . 23
- 4.3 Policy Interpretation . . . 24
  - 4.3.1 Provisions Service . . . 24
  - 4.3.2 Policy Decision Point . . . 25
  - 4.3.3 Policy Enforcement Point . . . 29
- 4.4 Print Operation . . . 30

5 **Related Work** 33

6 **Conclusion** 34

7 **Acknowledgments** 34

A **Proof of Concept: XML Documents** 34
- A.1 Data Types . . . 35
- A.2 Containers (State) . . . 36
- A.3 Discovery Operations . . . 36
- A.4 Operations on Elements . . . 37
- A.5 Operations on Attributes . . . 39
- A.6 Operations on Texts . . . 41
- A.7 Operations on Processing Instructions . . . 43
- A.8 Remark . . . 45

1 Introduction

Sensitive data is often protected by controlling access to its container. Two examples of containers are databases and file systems. Typically, databases are based on the relational model, whereas file systems are modeled as trees whose inner nodes are directories and whose leaves are files. For both examples, there are access-control models and systems (e.g., [3, 6, 11, 25]) that take the inner structure of the respective container into account and thus allow for fine-grained access control. This means that access is not granted or denied to a database or a file system as a whole, but rather to individual tables or rows of the database and to individual directories or files of the file system.

A third example of a data container is a document. When documents are protected by controlling access to the file system where they reside, users either have full access to a document or no access at all. However, in some contexts (cf. §2), fine-grained access control is also required for documents. There exist access-control models for a specific class of documents, namely Extensible Markup Language (XML) [29, 12, 20] documents (cf. §5). However, most of these models are limited to XML-encoded databases. Furthermore, the systems based on these models cannot protect data once it has been released to users.

What has been missing until now is an access-control system that is based on a fine-grained access-control model for documents, such as texts, spreadsheets, and presentations, and whose mechanisms not only enforce policies on a server but also on clients, both while data is within documents and in transit between documents. In this technical report[0], we present a formal model of such a system.

1.1 Contributions

We have defined a fine-grained model of a system for processing documents. As natural languages and semi-formal modeling languages like UML are not sufficiently precise, we have used the specification language Z [17] to define the states and operations of the system. Hence our first contribution is a formal model of an unprotected document-processing system.

Our second contribution is a policy language that allows users to formalize protection requirements that we have gathered for banking environments. Again we have taken a formal approach here and defined the policy language's abstract syntax in Z and its semantics (how access requests are evaluated) in a combination of (Object-)Z and the specification language Communicating Sequential Processes (CSP) [14] called CSP-OZ [8].

Our third contribution is to provide a foundation for controlling usage of documents. *Usage control* [23] is a notion that subsumes both server-side and client-side access control.[1] It is important in the document context as owners need assurance that the policies governing access to their documents are respected, even when other users incorporate parts of these documents in their own documents. To achieve this, we associate parts of each document with

[0] This TR is the full version of our ETRICS 2006 paper [24].

[1] Client-side access control is also called *rights management*. Note that enforcement requires the combination of classical access-control mechanisms with hardware-based or software-based rights-management mechanisms.

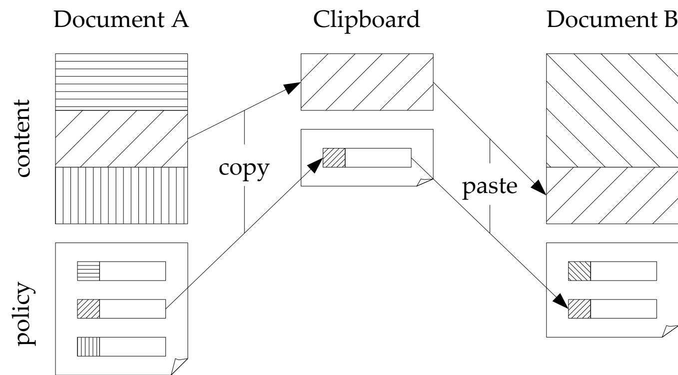

Figure 1: Sticky Policies

the respective parts of its policy and maintain this association over the document life-cycle. This amounts to a fine-grained variant of the sticky-policies paradigm [18, 16]: when content is copied (or cut) from a document to the clipboard and pasted into a document, then so is the respective part of the policy (cf. Figure 1).

1.2 Organization

In Section 2, we explain the context of this work and we derive high-level requirements from representative use cases. In Section 3, we introduce our document model and we formalize document content. In Section 4, we formalize document policies and their enforcement. In Section 5, we compare with related work, and we draw conclusions in Section 6.

2 Requirements

Documents take many forms and serve many purposes. In this technical report, we will restrict our focus to a setting common in the context of companies and other organizations. Stakeholders in these companies (i.e., users) create, exchange, read, and edit documents that contain security-sensitive data. In general, users cannot be trusted by the company because they may not understand the company's security requirements, they may be careless in their use of data, they may have an untrustworthy platform (e.g., compromised by a Trojan horse), or they may simply be untrustworthy themselves.

In what follows, we will assume that systems are trustworthy but users are not. That is, we focus on the functional security requirements for document processing necessary to handle careless or dishonest users. We will restrict our attention to the following representative use cases:

Annual Report: To mitigate the risk of insider trading, public access to the company's annual report must not be granted until a given date.

Company Guidelines: Before employees are granted access to data, they must

first acknowledge the company guidelines governing the handling of that data.

Presentation Slides: Presenters must be able to define different rules (policy) for different parts (content) within one and the same presentation document so that they need not create multiple differently censored versions.

From the first use case, we derive the requirement for *conditional access control*, i.e., for access-control decisions that depend on request parameters other than the subject, the object, and the action. From the second use case, we derive the requirement for *provisional access control*, i.e., for access-control decisions that depend on whether provisions have been made. From the third use case, we derive the requirement for *fine-grained access control*, i.e., for access-control models and systems where the objects protected are not the data containers (in our case documents) as a whole, but rather their constituent parts. We can derive the same requirements in other document-processing contexts. Examples include the review process of papers, the distribution of sample chapters of books, and the acceptance (or not) of end-user license agreements.

A notion related to *provisions* is *obligations*. Roughly speaking, provisions must hold when access is granted, while obligations must hold after access has been granted [4, 5, 13]. For example, a subject may be obliged not to disclose any information learned as a result of being granted access to some data. Since, by definition, this obligation cannot be enforced technically at the time of access, it is mapped to a provision which can be, namely the subject must have signed a non-disclosure agreement. Our model will not directly support obligations, but will support both conditions and provisions.

3 Document Content Model

In our model, documents are pairs consisting of a *content* component and a *policy* component. The content is where the data is stored and the policy describes what operations are allowed on both the content and the policy itself. Further components—which we do not model in this technical report—are either application-specific (e.g., a style sheet) or related to the security mechanisms (e.g., encrypted component-decryption keys). We model the content component in this section and the policy component in the next.

We have formalized our model in Z, which is a popular formal language based on typed set theory and first-order logic with equality. We have chosen Z as document processing is heavily data-oriented and Z is well-suited for data modeling. In particular, Z provides constructs for structuring and compositionally building data-oriented specifications: schemas are used to model the states of the system (*state schemas*) and operations on the state (*operation schemas*), and a *schema calculus* is provided to compose these subspecifications.

3.1 Content Model

Many kinds of content are structured hierarchically. For example, a book consists of chapters, sections, and paragraphs. To reflect this, we model content as a rooted tree whose nodes have attributes (i.e., name-value pairs). This model

is quite general and we can easily specialize it not only to different document formats, like XML (cf. Appendix A), but also to directory information bases and to file systems (cf. §3.2).

3.1.1 Data Types and Auxiliary Functions

We introduce four data types: *Name* and *Value* are basic types. *Name* represents the set of attribute names and *Value* the set of attribute values. As we shall see, security-sensitive data is stored as attribute values. *Attributes* is the set of finite sets of name-value pairs in which a name maps to at most one value, i.e., members of this type are functions mapping finitely many names to values. Finally, *Tree* is a recursive type where each node has attributes and a sequence of subtrees. In Z, we express all this as follows:

$$[Name, Value]$$
$$Attributes == Name \ffun Value$$
$$Tree ::= Node\langle\!\langle Attributes \times \operatorname{seq} Tree \rangle\!\rangle$$

We have specified several auxiliary functions on these data types, which we will use below:
Given a tree, the following function returns the set of all valid paths in the tree.

$$
\begin{array}{|l}
TreeDomainF : Tree \rightarrow \mathbb{P} \operatorname{seq} \mathbb{N}_1 \\
\hline
\forall a : Attributes;\ ts : \operatorname{seq} Tree \bullet \\
\quad TreeDomainF(Node(a, ts)) = \{\langle\rangle\} \cup \{i : \mathbb{N}_1;\ p : \operatorname{seq} \mathbb{N}_1 \mid \\
\quad i \leq \#ts \land p \in TreeDomainF(ts(i)) \bullet \langle i \rangle \frown p\}
\end{array}
$$

Given a tree and a valid path, the following function returns the subtree at (the end of) the path.

$$
\begin{array}{|l}
ReadNodeF : Tree \times \operatorname{seq} \mathbb{N}_1 \pfun Tree \\
\hline
\forall t : Tree \bullet ReadNodeF(t, \langle\rangle) = t \\
\forall p : \operatorname{seq}_1 \mathbb{N}_1;\ a : Attributes;\ ts : \operatorname{seq} Tree \mid \\
\quad first(p) \leq \#ts \bullet ReadNodeF(Node(a, ts), p) = ReadNodeF(ts(first(p)), tail(p))
\end{array}
$$

Given a tree and a valid path, the following function returns the attributes of the root node of the subtree at the path.

$$
\begin{array}{|l}
ReadAttributesF : Tree \times \operatorname{seq} \mathbb{N}_1 \pfun Attributes \\
\hline
\forall t : Tree;\ p : \operatorname{seq} \mathbb{N}_1;\ a : Attributes;\ ts : \operatorname{seq} Tree \mid \\
\quad ReadNodeF(t, p) = Node(a, ts) \bullet ReadAttributesF(t, p) = a
\end{array}
$$

Given two trees and a path, the following function returns the tree that results from adding the second tree to the first tree at the path.

$$
\begin{array}{|l}
AddNodeF : Tree \times \mathrm{seq}_1\, \mathbb{N}_1 \times Tree \rightarrowtail Tree \\
\hline
\forall p : \mathrm{seq}_1\, \mathbb{N}_1;\ t : Tree;\ a : Attributes;\ ts : \mathrm{seq}\, Tree \mid \\
\quad front(p) \in TreeDomainF(Node(a, ts)) \wedge last(p) \leq \#ts + 1 \bullet \\
\quad AddNodeF(Node(a, ts), p, t) = \mathbf{if}\ \#p = 1\ \mathbf{then} \\
\quad Node(a, (1 \mathinner{..} p(1) - 1) \upharpoonright ts \frown \langle t \rangle \frown (p(1) \mathinner{..} \#ts) \upharpoonright ts)\ \mathbf{else} \\
\quad Node(a, (1 \mathinner{..} p(1) - 1) \upharpoonright ts \frown \langle AddNodeF(ts(first(p)), tail(p), t) \rangle \frown (p(1) \mathinner{..} \#ts) \upharpoonright ts)
\end{array}
$$

The following function returns the absolute value of its integer argument.

$$
\begin{array}{|l}
AbsValF : \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
\forall z : \mathbb{Z} \mid z < 0 \bullet AbsValF(z) = -z \\
\forall z : \mathbb{Z} \mid z \geq 0 \bullet AbsValF(z) = z
\end{array}
$$

3.1.2 Containers (State) and Initialization Operations

The state of documents and the clipboard are represented as schemas. As mentioned in Section 1.1, making the clipboard and the clipboard-related operations part of the model is a prerequisite for fine-grained *sticky policies*. The first schema below represents the content component of documents. The second schema represents the content component of the clipboard. Both consist of a single binding stating that the document content is a tree called *root* and the clipboard content is a tree called *cCache*.

$$
\begin{array}{|l}
\text{--- } DocumentContent \text{ ---} \\
root : Tree \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\text{--- } ClipboardContent \text{ ---} \\
cCache : Tree \\
\hline
\end{array}
$$

In general, each state schema comes with an initialization schema that specifies the initial state and establishes the state invariants. In the case of document content and clipboard content, the empty node is assigned to the root and the cache.

$$
\begin{array}{|l}
\text{--- } InitDocumentContent \text{ ---} \\
DocumentContent \\
\hline
root = Node(\{\}, \langle\rangle) \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\text{--- } InitClipboardContent \text{ ---} \\
Clipboard \\
\hline
cCache = Node(\{\}, \langle\rangle) \\
\hline
\end{array}
$$

The *InitDocumentContent* operation is the first operation to be performed when a new document is created. Since the policies are sticky, performing the *InitClipboardContent* operation is only necessary to set the clipboard into a correct state (e.g., when the document processor is started up), not for security reasons.

3.1.3 Operations

We have defined more than a dozen operations, most of which change the state of the document, the clipboard, or both. They are reading, adding, deleting, copying, cutting, and pasting a node as well as reading, adding, deleting, changing, copying, cutting, and pasting an attribute. Reading a node is special in that it returns the names of its attributes and the number of its children, but not any value. There is no change-node operation. Note that our convention is that schemas whose names end in *C* or *P* specify a content-related operation and a policy-related operation, respectively.

Reading a node means discovering its structure (i.e., the names of its attributes—but not their values—and its number of children).[2] The precondition is that the path given is valid. The number of children is determined by taking the last element of the rightmost child's path.

$$
\begin{array}{l}
\textit{ReadNodeC} \\
\hline
\Xi DocumentContent \\
path? : \mathrm{seq}\,\mathbb{N}_1 \\
attributesDom! : \mathbb{P}\,Name \\
childrenNr! : \mathbb{N} \\
\hline
path? \in TreeDomainF(root) \\
attributesDom! = \mathsf{dom}\,ReadAttributesF(root, path?) \\
childrenNr! = max(\{0\} \cup \{n : \mathbb{N}_1 \mid path? \frown \langle n \rangle \in TreeDomainF(root)\}) \\
\hline
\end{array}
$$

Adding a node means not just adding an empty node, but a subtree. The preconditions are that the parent exists and that the new child will either be the first child or that all children left of it exist.

$$
\begin{array}{l}
\textit{AddNodeC} \\
\hline
\Delta DocumentContent \\
path? : \mathrm{seq}_1\,\mathbb{N}_1 \\
attributes? : Attributes \\
treeSequence? : \mathrm{seq}\,Tree \\
\hline
front(path?) \in TreeDomainF(root) \\
last(path?) = 1 \lor front(path?) \frown \langle last(path?) - 1 \rangle \in TreeDomainF(root) \\
root' = AddNodeF(root, path?, Node(attributes?, treeSequence?)) \\
\hline
\end{array}
$$

Deleting a node means deleting the subtree rooted at that node. Note that we take advantage of the *DeleteNodeC* operation basically being the inverse of the *AddNodeC* operation. Nevertheless, we must explicitly specify the precondition, namely that the path given is valid.

$$
\begin{array}{l}
\textit{DeleteNodeC} \\
\hline
\Delta DocumentContent \\
path? : \mathrm{seq}_1\,\mathbb{N}_1 \\
\hline
path? \in TreeDomainF(root) \\
\exists\, attributes? : Attributes;\ treeSequence? : \mathrm{seq}\,Tree \bullet \\
\quad AddNodeC[root', root/root, root'] \\
\hline
\end{array}
$$

Copying a node results in the subtree rooted at that node being cached in the clipboard. The precondition is that the path given is valid. The copy semantics is *by value* (i.e., not by reference).

[2]Unlike Stoica and Farkas [26] and Gabillon [9], we shall not support hiding the existence of attributes or children of nodes a subject is authorized to read.

$$
\begin{array}{|l}
\hline
\textit{CopyNodeC} \\
\Xi \textit{DocumentContent} \\
\Delta \textit{ClipboardContent} \\
\textit{path}? : \operatorname{seq} \mathbb{N}_1 \\
\hline
\textit{path}? \in \textit{TreeDomainF}(\textit{root}) \\
\textit{cCache}' = \textit{ReadNodeF}(\textit{root}, \textit{path}?) \\
\hline
\end{array}
$$

Cutting a node means copying the node followed by deleting it. Note that because of *DeleteNodeC*'s precondition, the sequence *path*? must not be empty.

$$\textit{CutNodeC} \mathrel{\widehat=} \textit{CopyNodeC} \mathbin{;} \textit{DeleteNodeC}$$

Pasting a node means adding the tree cached in the clipboard. The preconditions are implicit in the first conjunct and in the *AddNodeC* operation.

$$
\begin{array}{|l}
\hline
\textit{PasteNodeC} \\
\Delta \textit{DocumentContent} \\
\Xi \textit{ClipboardContent} \\
\textit{path}? : \operatorname{seq}_1 \mathbb{N}_1 \\
\hline
\exists a : \textit{Attributes};\ ts : \operatorname{seq} \textit{Tree} \bullet \textit{cCache} = \textit{Node}(a, ts) \land \\
\quad \textit{AddNodeC}[a/\textit{attributes}?, ts/\textit{treeSequence}?] \\
\hline
\end{array}
$$

Reading an attribute means reading its value. The preconditions are that the path given is valid and that the name is in the domain of the respective node's attributes.

$$
\begin{array}{|l}
\hline
\textit{ReadAttributeC} \\
\Xi \textit{DocumentContent} \\
\textit{path}? : \operatorname{seq} \mathbb{N}_1 \\
\textit{name}? : \textit{Name} \\
\textit{value}! : \textit{Value} \\
\hline
\textit{path}? \in \textit{TreeDomainF}(\textit{root}) \\
\textit{name}? \in \operatorname{dom} \textit{ReadAttributesF}(\textit{root}, \textit{path}?) \\
\exists a : \textit{Attributes};\ ts : \operatorname{seq} \textit{Tree} \mid \\
\quad \textit{ReadNodeF}(\textit{root}, \textit{path}?) = \textit{Node}(a, ts) \bullet (\textit{name}?, \textit{value}!) \in a \\
\hline
\end{array}
$$

Adding an attribute means adding a value under a new name. The preconditions are that the path given is valid and that the name is not already in the domain of the respective node's attributes. Note that the operation neither changes the tree structure nor the attributes of the other nodes.

AddAttributeC

$\Delta DocumentContent$
$path? : \mathrm{seq}\,\mathbb{N}_1$
$name? : Name$
$value? : Value$

$path? \in TreeDomainF(root)$
$name? \notin \mathsf{dom}\, ReadAttributesF(root, path?)$
$TreeDomainF(root') = TreeDomainF(root)$
$\forall p : \mathrm{seq}\,\mathbb{N}_1 \mid p \in TreeDomainF(root) \land p \neq path? \bullet$
$\quad ReadAttributesF(root', p) = ReadAtrributesF(root, p)$
$ReadAttributesF(root', path?) = ReadAttributesF(root, path?) \cup (name? \mapsto value?)$

Deleting an attribute means deleting the name-value pair. Note that we take advantage of the *DeleteAttributeC* operation basically being the inverse of the *AddAttributeC* operation. Note further that one of *AddAttributeC*'s preconditions ($name? \notin \mathsf{dom}\, ReadAttributesF(root, path?)$) becomes one of *DeleteAttributeC*'s postconditions ($name? \notin \mathsf{dom}\, ReadAttributesF(root', path?)$) and one of *AddAttributeC*'s postconditions ($ReadAttributesF(root', path?) = ReadAttributesF(root, path?) \cup (name? \mapsto value?)$) becomes one of *DeleteAttributeC*'s preconditions ($ReadAttributesF(root, path?) = ReadAttributesF(root', path?) \cup (name? \mapsto value?)$).

DeleteAttributeC

$\Delta DocumentContent$
$path? : \mathrm{seq}\,\mathbb{N}_1$
$name? : Name$

$\exists\, value? : Value \bullet AddAttributeC[root', root / root, root']$

Changing an attribute means deleting the attribute followed by adding an attribute with the name of the deleted attribute.

$ChangeAttributeC \mathrel{\widehat{=}} DeleteAttributeC \fatsemi AddAttributeC$

Copying an attribute results in the attribute value being cached in the clipboard under its current name, the latter being irrelevant though (cf. *PasteAttributeC*).

CopyAttributeC

$\Xi DocumentContent$
$\Delta ClipboardContent$
$path? : \mathrm{seq}\,\mathbb{N}_1$
$name? : Name$

$\exists\, value! : Value \bullet ReadAttributeC \land cCache' = Node(\{name? \mapsto value!\}, \langle\rangle)$

Cutting an attribute means copying the attribute followed by deleting it.

$CutAttributeC \mathrel{\widehat{=}} CopyAttributeC \fatsemi DeleteAttributeC$

Pasting an attribute means adding the attribute value cached in the clip-

board under a new name. The preconditions are implicit in the first conjunct (exactly one attribute is cached in the clipboard) and in the *AddAttributeC* operation.

$$
\begin{array}{l}
\textit{PasteAttributeC} \\
\hline
\Delta \textit{DocumentContent} \\
\Xi \textit{ClipboardContent} \\
\textit{path}? : \mathrm{seq}\, \mathbb{N}_1 \\
\textit{name}? : \textit{Name} \\
\hline
\exists\, n : \textit{Name};\ v : \textit{Value} \bullet \textit{cCache} = \textit{Node}(\{n \mapsto v\}, \langle\rangle) \land \\
\quad \textit{AddAttributeC}[v/\textit{value}?]
\end{array}
$$

3.2 Example Refinements

In Section 1.1, we claimed that our content model encompasses containers other than documents. In this subsection, we support this claim by refining the above-defined data types and state schemas to a generic file system on the one hand and to a Lightweight Directory Access Protocol (LDAP)-compliant directory information base on the other hand. Unlike the proof-of-concept refinement to XML documents (cf. Appendix A), we provide these two refinements for illustrative purposes only and therefore refrain from refining the operations as well.

3.2.1 File System

Two kinds of attributes found in file systems are name attributes (whose value is the name of the directory or file) and value attributes (whose value is the content of the file).

$$
\begin{array}{l}
\textit{nameName}, \textit{valueName} : \textit{Name} \\
\hline
\textit{nameName} \neq \textit{valueName}
\end{array}
$$

Directories have a name, no value, further attributes (e.g., the time of creation), and children. Files have a name, a value, further attributes (e.g., the MIME type), and no children.

$$
\begin{array}{l}
\textit{Directory} == \{t : \textit{Tree} \mid (\exists\, a : \textit{Attributes};\ \textit{ts} : \mathrm{seq}\, \textit{Tree} \mid \\
\quad \textit{nameName} \in \mathrm{dom}\, a \land \textit{valueName} \notin \mathrm{dom}\, a \bullet t = \textit{Node}(a, \textit{ts}))\} \\
\textit{File} == \{t : \textit{Tree} \mid (\exists\, a : \textit{Attributes} \mid \\
\quad \textit{nameName} \in \mathrm{dom}\, a \land \textit{valueName} \in \mathrm{dom}\, a \bullet t = \textit{Node}(a, \langle\rangle))\}
\end{array}
$$

The above definition of directories still allows for them to have children other than directories or files. In a file system, however, we only find directories or files.

$$
\begin{array}{|l}
\hline
\textit{FileSystem} \\
\hline
\textit{DocumentContent} \\
\hline
\textit{root} \in \textit{Directory} \\
\forall\, p : \textit{TreeDomainF}(\textit{root}) \mid p \neq \langle\rangle \bullet \\
\quad \textit{ReadNodeF}(\textit{root}, p) \in \textit{Directory} \lor \textit{ReadNodeF}(\textit{root}, p) \in \textit{File} \\
\hline
\end{array}
$$

Of the operations we defined for documents, even copy, cut, and paste make sense for file systems, either to duplicate a directory or file within the same file system or to check it out into another. For obvious reasons, the document model lacks an execute operation; however, if the file system represents a revision control system such as Subversion[3] or CVS[4], none is needed.

3.2.2 LDAP Directory

One kind of attributes found in directory services are name attributes (whose value is the name of the entry).

$$
\mid \textit{RDN} : \textit{Name}
$$

Entries have a relative distinguished name (RDN), further attributes with one or more values, and children.

$$
\begin{array}{l}
[\textit{SValue}] \\
\textit{Value} == \mathbb{P}_1\, \textit{SValue} \\
\textit{Entry} == \{ t : \textit{Tree} \mid (\exists\, a : \textit{Attributes};\ ts : \operatorname{seq} \textit{Tree} \mid \\
\quad \textit{RDN} \in \operatorname{dom} a \bullet t = \textit{Node}(a, ts)) \}
\end{array}
$$

The above definition of entries still allows for them to have children other than entries. In a directory information base, however, we only find entries.

$$
\begin{array}{|l}
\hline
\textit{DirectoryInformationBase} \\
\hline
\textit{DocumentContent} \\
\hline
\forall\, p : \textit{TreeDomainF}(\textit{root}) \bullet \textit{ReadNodeF}(\textit{root}, p) \in \textit{Entry} \\
\hline
\end{array}
$$

4 Document Policy Model

In this section we present our policy language and access-control architecture. We have designed the language to meet our domain-specific requirements for controlling access to document content (as just modeled). Our architecture is an adaptation of the XACML data-flow model [22]. We first present these ideas informally and afterwards present the formal specification.

[3] http://subversion.tigris.org/
[4] http://www.nongnu.org/cvs/

4.1 Informal Description

4.1.1 Policy Language

Our access-control model is role-based, where policies express relations between roles and permissions and where subjects are *users* acting in a *role*. Additionally, policies incorporate a concept of ownership adapted from discretionary access control (DAC), where every object has an owner, namely the user that created it. Users are allowed all forms of access to objects they own and can arbitrarily add and delete role-permission assignments for these objects as well. However, unlike DAC models, our model does not leave to a subject's discretion any data the subject has (read) access to.

Permissions relate *objects* with *actions* (not to be confused with operations) that are further constrained by *conditions* and *provisions*. That is, permissions only apply when the condition evaluates to *true* in the current *environment*. As their name implies, permissions always *grant* access. Nevertheless, grants are tentative until the provisions have been made. By design, conflicts (i.e., different sets of provisions) cannot arise from more than one permission applying to a request and as a result there is no need for conflict-resolution strategies. Subjects can be permitted to delegate their reading and editing permissions to other subjects.

We limit ourselves to a single editing action, which we call *change*. This is in contrast to other models (cf. §5), which typically have the actions add, delete, and update (when integrity is a concern). In our model, to add or delete a child node or an attribute, a subject must be allowed to change the parent.

Table 1: Operations permitted by Actions

Object / Action	**attribute**	**node**
change	*not applicable*	add attribute to node add child node to node delete an attribute from node delete subtree rooted at a child node change an attribute's value
add	*not applicable*	add attribute to node add child node to node
delete	delete attribute	delete subtree rooted at node
update	change attribute value	*not applicable*

Let us discuss the change action in more detail. Table 1 lists the operations permitted by our change action and by the usual add, delete, and update actions. We claim that giving add, delete, and update permissions individually is unsatisfactory, in particular in the context of document editing which requires the ability to undo operations. Suppose, for example, that a subject has the permission to add an object to a node. Undoing adding an attribute or a node could be supported by giving the subject the permission to delete the attribute or the node. However, the permissions required to undo pasting an attribute or a node (i.e., a subtree) are less clear. Similarly, does deleting a node require the permission to delete all descendants of the node? And does deleting an

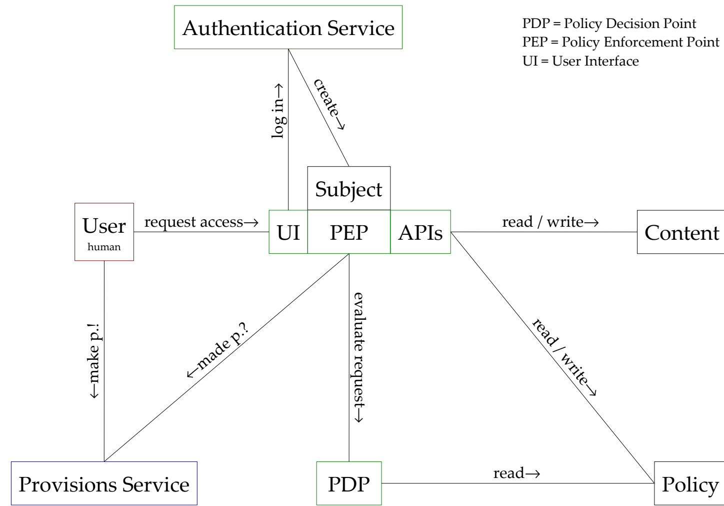

Figure 2: Access-Control Actors

object require the permission to read the object? Now suppose that a subject has the permission to delete an object but not the permission to add an object to the object's parent node. Can deleting the object be undone without giving the subject "new" (add) permissions on "old" nodes? We avoid these questions with our approach and instead provide users with a simple semantics of an editing action whose consequences are easy to understand: a subtree can either be changed in arbitrary ways or not at all. Note that this simplification has no negative effects when confidentiality is the main security goal, as it is in the use cases in Section 2.

4.1.2 Access-Control Architecture

The architecture of our system is an adaptation of the XACML data-flow model [22] and is shown in Figure 2. The system runs on a User's client. Content and Policy are components of a document that the user has opened with the system. When several documents are open, only one is currently active. After successful login, the user is represented by a Subject and accesses documents through the user interface (UI). If security were not a concern, there would be no policy and the UI would directly access the document application programming interfaces (APIs)[5] when reading and writing content. However, since security is indeed a concern, the UI accesses the document APIs via the policy enforcement point (PEP). The PEP grants or denies access (i.e., blocks operations) based on access decisions made by the Policy Decision Point (PDP) and based on whether certain provisions have been made, for which the PEP consults the Provisions Service. That is, the PEP mediates access while the PDP makes (tentative) access decisions.

[5]In the context of XML documents, the Document Object Model (DOM) [28] is a well-known example for such an API.

4.2 Formal Specification

4.2.1 Data Types

The most elementary concepts of any access-control model are subjects, objects, and actions. In our model, subjects are pairs consisting of a role ID and a user ID. The set of all role IDs and the set of all user IDs are basic types of the specification. Objects are nodes or attributes at a given path. The ability to speak about nodes and attributes at arbitrary positions in the document content is straightforward, given our content model, and allows for fine-grained access control. We have defined four actions: *read*, *change*, *print*, and *delegate*.

$$[\mathit{RoleID}, \mathit{UserID}]$$
$$\mathit{Subject} == \mathit{RoleID} \times \mathit{UserID}$$
$$\mathit{Object} ::= \mathit{Node}\langle\langle \operatorname{seq} \mathbb{N}_1 \rangle\rangle \mid \mathit{Attribute}\langle\langle \operatorname{seq} \mathbb{N}_1 \times \mathit{Name} \rangle\rangle$$
$$\mathit{Action} ::= \mathit{read} \mid \mathit{change} \mid \mathit{print} \mid \mathit{delegate}$$

Permissions with *read* actions grant read operations. Permissions with *print* actions grant the print operation (cf. §4.4). Permissions with *delegate* actions grant subjects the operation which delegates their *read*- and *change*-based permissions to other subjects (i.e., the permission to print or delegate cannot be delegated). The permissions with *change* actions grant add and delete (and thus change, cut, and paste) operations. The prerequisite for granting any operation is that a read operation would be granted, too.

The applicability of permissions can be limited by conditions on the environment and user. For now, *time* is the only environmental property modeled. To satisfy a condition, the request environment must be in the condition's set of environments and the requesting user must be in the condition's set of users. Time-dependent permissions are motivated by the use cases given in Section 2. User-dependent permissions are necessary for delegation. For example, if a manager wants to delegate some of her permissions to her secretary (but not to all employees whose role is secretary) she can make the delegated permissions depend on the secretary's user ID.

$$\mathit{Environment} ::= \mathit{Timestamp}\langle\langle \mathbb{Z} \rangle\rangle$$
$$\mathit{Condition} == \mathbb{P}\,\mathit{Environment} \times \mathbb{P}\,\mathit{UserID}$$
$$\mathit{AnyTime} == \{t : \mathbb{Z} \bullet \mathit{Timestamp}(t)\}$$
$$\mathit{AnyUser} == \mathit{UserID}$$

We support two kinds of provisions: *log* and *sign*.[6] When a permission depends on a log provision, the system providing access must log the message specified in the provision before access can be granted. When a permission depends on a sign provision, the user requesting access must have signed the agreement specified in the provision before access can be granted. Examples for agreements are end-user license agreements and non-disclosure agreements as well as company guidelines. The set of all messages and the set of all agreement IDs are basic types of the specification.

[6] These two provisions are essential given our requirements, but others could be added simply by extending the *Provision* data type.

$[Message, AgreementID]$
$Provision ::= Log\langle\langle Message\rangle\rangle \mid Sign\langle\langle AgreementID\rangle\rangle$

Now we have specified all components of permission tuples. Given the request parameters, the PDP checks whether a permission matches the request. If so, the PDP responds with *Grant* and the permission's set of provisions and if not, the PDP responds with *Deny*.

$Permission == Object \times Action \times Condition \times \mathbb{P}\, Provision$
$Request == Subject \times Object \times Action \times Environment$
$Response ::= Grant\langle\langle \mathbb{P}\, Provision\rangle\rangle \mid Deny$

Figure 3 depicts the relations between some of the types just specified.

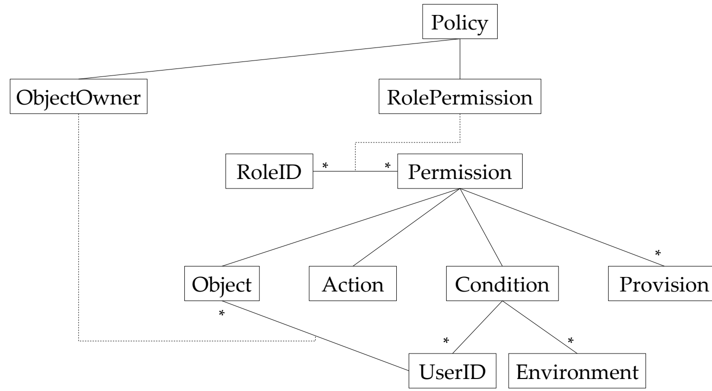

Figure 3: Policy Meta-Model

4.2.2 Containers (State) and Initialization Operations

The first schema shown below specifies what constitutes the policy component for documents. Its predicate states that permissions with the action *change* must have a node as object and that at most one permission can match a given request. The second schema specifies what constitutes the policy component for the clipboard. Both schemas declare variables of the same type: a partial, finite function that maps objects to their owner and the role-permission relation for role-based access control.

DocumentPolicy

$ObjectOwner : Object \mathrel{+\!\!\!\!\twoheadrightarrow} UserID$
$RolePermission : RoleID \leftrightarrow Permission$

$\forall r : RoleID;\ p : Permission \mid r \mapsto p \in RolePermission \bullet$
$\quad p.2 \neq change \lor (\exists path : \operatorname{seq} \mathbb{N}_1 \bullet p.1 = Node(path))$
$\forall r_1, r_2 : RoleID;\ p_1, p_2 : Permission \mid r_1 \mapsto p_1 \in RolePermission \land$
$\quad r_2 \mapsto p_2 \in RolePermission \bullet r_1 \neq r_2 \lor p_1 = p_2 \lor$
$\quad p_1.1 \neq p_2.1 \lor p_1.2 \neq p_2.2 \lor$
$\quad first(p_1.3) \cap first(p_2.3) = \varnothing \lor second(p_1.3) \cap second(p_2.3) = \varnothing$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ClipboardPolicy}} \\
\hline
ooCache : Object \twoheadrightarrow\!\!\!\!\!\!\!\!+\ UserID \\
rpCache : RoleID \leftrightarrow Permission \\
\hline
\end{array}$$

Using Z's schema calculus, here schema conjunction, we can now formally express that both documents and the clipboard are pairs consisting of a content component and a policy component (cf. the pairs depicted in Figure 1).

$Document \mathrel{\widehat{=}} DocumentContent \land DocumentPolicy$
$Clipboard \mathrel{\widehat{=}} ClipboardContent \land ClipboardPolicy$

The *InitDocumentPolicy* is the first operation to be performed when a new document is created. Since the policies are sticky, performing the *InitClipboard-Policy* operation is only necessary to bring the clipboard into a correct state (e.g., when the document processor is started up), not for security reasons.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitDocumentPolicy}} \\
\hline
DocumentPolicy \\
subject? : Subject \\
\hline
ObjectOwner = \{Node(\langle\rangle) \mapsto second(subject?)\} \\
RolePermission = \{\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitClipboardPolicy}} \\
\hline
ClipboardPolicy \\
\hline
ooCache = \{\} \\
rpCache = \{\} \\
\hline
\end{array}$$

Initializing a document and the clipboard are transactions operating simultaneously on both the content and policy components.

$InitDocument \mathrel{\widehat{=}} InitDocumentContent \land InitDocumentPolicy$
$InitClipboard \mathrel{\widehat{=}} InitClipboardContent \land InitClipboardPolicy$

4.2.3 Operations on Nodes and Auxiliary Functions

Reading a node has no effect on the document's policy.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadNodeP}} \\
\hline
\Xi DocumentPolicy \\
\hline
\end{array}$$

Adding a node results in the user adding the node becoming the owner of the node as well as of all its descendants and in potentially shifting existing object addresses. To reduce the complexity of the *AddNodeP* schema, we have specified three auxiliary functions:
Given a tree and a valid path within that tree, the following function returns the children of the root node of the subtree at that path.

$$
\begin{array}{l}
ReadChildrenF : Tree \times \operatorname{seq} \mathbb{N}_1 \nrightarrow \operatorname{seq} Tree \\
\hline
\forall t : Tree;\ p : \operatorname{seq} \mathbb{N}_1;\ a : Attributes;\ ts : \operatorname{seq} Tree \mid \\
\quad ReadNodeF(t, p) = Node(a, ts) \bullet ReadChildrenF(t, p) = ts
\end{array}
$$

Given the ID of the user who adds the node, the address of where the node is being added, the node's attributes, and the node's children, the following function returns ownership information to be added to the policy. That is, the set of all pairs whose first element refers to the node or one of its descendants and whose second element is the user ID is returned.

$$
\begin{array}{l}
MakeOOsF : UserID \times \operatorname{seq}_1 \mathbb{N}_1 \times Attributes \times \operatorname{seq} Tree \rightarrow \mathbb{P}(Object \times UserID) \\
\hline
\forall u : UserID;\ p_1 : \operatorname{seq}_1 \mathbb{N}_1;\ a : Attributes;\ ts : \operatorname{seq} Tree \bullet \\
\quad MakeOOsF(u, p_1, a, ts) = \{Node(p_1) \mapsto u\} \cup \{n : \operatorname{dom} a \bullet \\
\quad Attribute(p_1, n) \mapsto u\} \cup (\bigcup \{i : \mathbb{N}_1 \mid i \leq \#ts \bullet MakeOOsF(u, p_1 \frown \langle i \rangle, \\
\quad ReadAttributesF(ts(i), \langle\rangle), ReadChildrenF(ts(i), \langle\rangle))\})
\end{array}
$$

Given an object (i.e., a reference in the policy to a node or an attribute in the content) and the address of where the node is being added, the following function returns the object that references the same node or attribute in the tree with the added node.

$$
\begin{array}{l}
ShiftAddObjectF : Object \times \operatorname{seq}_1 \mathbb{N}_1 \rightarrow Object \\
\hline
\forall p : \operatorname{seq} \mathbb{N}_1;\ p_1 : \operatorname{seq}_1 \mathbb{N}_1 \bullet ShiftAddObjectF(Node(p), p_1) = \\
\quad \textbf{if}\ front(p_1)\ prefix\ p \land \#p_1 \leq \#p \land last(p_1) \leq p(\#p_1)\ \textbf{then} \\
\quad Node(front(p_1) \frown \langle p(\#p_1) + 1 \rangle \frown (\#p_1 + 1 \mathinner{..} \#p) \upharpoonright p)\ \textbf{else} \\
\quad Node(p) \\
\forall p : \operatorname{seq} \mathbb{N}_1;\ n : Name;\ p_1 : \operatorname{seq}_1 \mathbb{N}_1 \bullet ShiftAddObjectF(Attribute(p, n), p_1) = \\
\quad \textbf{if}\ front(p_1)\ prefix\ p \land \#p_1 \leq \#p \land last(p_1) \leq p(\#p_1)\ \textbf{then} \\
\quad Attribute(front(p_1) \frown \langle p(\#p_1) + 1 \rangle \frown (\#p_1 + 1 \mathinner{..} \#p) \upharpoonright p, n)\ \textbf{else} \\
\quad Attribute(p, n)
\end{array}
$$

$$
\begin{array}{l}
\text{--- } AddNodeP \text{ ---} \\
\Delta DocumentPolicy \\
subject? : Subject \\
path? : \operatorname{seq}_1 \mathbb{N}_1 \\
attributes? : Attributes \\
treeSequence? : \operatorname{seq} Tree \\
\hline
ObjectOwner' = \{o : Object;\ u : UserID \mid o \mapsto u \in ObjectOwner \bullet \\
\quad ShiftAddObjectF(o, path?) \mapsto u\} \cup MakeOOsF(second(subject?), path?, \\
\quad attributes?, treeSequence?) \\
RolePermission' = \{r : RoleID;\ o : Object;\ a : Action;\ c : Condition; \\
\quad ps : \mathbb{P}\, Provision \mid r \mapsto (o, a, c, ps) \in RolePermission \bullet \\
\quad r \mapsto (ShiftAddObjectF(o, path?), a, c, ps)\}
\end{array}
$$

Deleting a node results in all ownership information and all permissions related to the node as well as to all its descendants being removed and in existing object addresses potentially being shifted. To reduce the complexity of the

DeleteNodeP schema, we have specified one auxiliary function:

Given an object (i.e., a reference in the policy to a node or an attribute in the content) and the address of where the node is being deleted from, the following function returns the object that references the same node or attribute in the tree with the deleted node.

$$
\begin{array}{l}
\mathit{ShiftDeleteObjectF} : \mathit{Object} \times \mathrm{seq}_1\, \mathbb{N}_1 \rightarrow \mathit{Object} \\
\hline
\forall p : \mathrm{seq}\, \mathbb{N}_1;\ p_1 : \mathrm{seq}_1\, \mathbb{N}_1 \bullet \mathit{ShiftDeleteObjectF}(\mathit{Node}(p), p_1) = \\
\quad \mathbf{if}\ \mathit{front}(p_1)\ \mathit{prefix}\ p \wedge \#p_1 \leq \#p \wedge \mathit{last}(p_1) \leq p(\#p_1)\ \mathbf{then} \\
\quad \mathit{Node}(\mathit{front}(p_1) \frown \langle p(\#p_1) - 1 \rangle \frown (\#p_1 + 1 \mathrel{..} \#p) \upharpoonright p)\ \mathbf{else} \\
\quad \mathit{Node}(p) \\
\forall p : \mathrm{seq}\, \mathbb{N}_1;\ n : \mathit{Name};\ p_1 : \mathrm{seq}_1\, \mathbb{N}_1 \bullet \mathit{ShiftDeleteObjectF}(\mathit{Attribute}(p, n), p_1) = \\
\quad \mathbf{if}\ \mathit{front}(p_1)\ \mathit{prefix}\ p \wedge \#p_1 \leq \#p \wedge \mathit{last}(p_1) \leq p(\#p_1)\ \mathbf{then} \\
\quad \mathit{Attribute}(\mathit{front}(p_1) \frown \langle p(\#p_1) - 1 \rangle \frown (\#p_1 + 1 \mathrel{..} \#p) \upharpoonright p, n)\ \mathbf{else} \\
\quad \mathit{Attribute}(p, n)
\end{array}
$$

$$
\begin{array}{l}
\mathit{DeleteNodeP} \\
\hline
\Delta \mathit{DocumentPolicy} \\
\mathit{path?} : \mathrm{seq}_1\, \mathbb{N}_1 \\
\hline
\mathit{ObjectOwner}' = \{ p : \mathrm{seq}\, \mathbb{N}_1;\ u : \mathit{UserID} \mid \\
\quad \mathit{Node}(p) \mapsto u \in \mathit{ObjectOwner} \wedge \neg\ (\mathit{path?}\ \mathit{prefix}\ p) \bullet \\
\quad \mathit{ShiftDeleteObjectF}(\mathit{Node}(p), \mathit{path?}) \mapsto u \} \cup \{ p : \mathrm{seq}\, \mathbb{N}_1;\ n : \mathit{Name}; \\
\quad u : \mathit{UserID} \mid \mathit{Attribute}(p, n) \mapsto u \in \mathit{ObjectOwner} \wedge \neg\ (\mathit{path?}\ \mathit{prefix}\ p) \bullet \\
\quad \mathit{ShiftDeleteObjectF}(\mathit{Attribute}(p, n), \mathit{path?}) \mapsto u \} \\
\mathit{RolePermission}' = \{ r : \mathit{RoleID};\ p : \mathrm{seq}\, \mathbb{N}_1;\ a : \mathit{Action};\ c : \mathit{Condition}; \\
\quad ps : \mathbb{P}\, \mathit{Provision} \mid r \mapsto (\mathit{Node}(p), a, c, ps) \in \mathit{RolePermission} \wedge \\
\quad \neg\ (\mathit{path?}\ \mathit{prefix}\ p) \bullet r \mapsto (\mathit{ShiftDeleteObjectF}(\mathit{Node}(p), \mathit{path?}), a, c, ps) \} \cup \\
\quad \{ r : \mathit{RoleID};\ p : \mathrm{seq}\, \mathbb{N}_1;\ n : \mathit{Name};\ a : \mathit{Action};\ c : \mathit{Condition};\ ps : \mathbb{P}\, \mathit{Provision} \mid \\
\quad r \mapsto (\mathit{Attribute}(p, n), a, c, ps) \in \mathit{RolePermission} \wedge \neg\ (\mathit{path?}\ \mathit{prefix}\ p) \bullet \\
\quad r \mapsto (\mathit{ShiftDeleteObjectF}(\mathit{Attribute}(p, n), \mathit{path?}), a, c, ps) \}
\end{array}
$$

Copying a node changes the clipboard policy but not the document policy. As specified in *CopyNodeC* on page 8, the subtree being copied becomes the tree in the clipboard content. Therefore, the common path prefix (*path?*) is removed in the course of copying those parts of the document policy related to the subtree being copied.

$$
\begin{array}{l}
\textit{CopyNodeP} \\
\hline
\Xi \textit{DocumentPolicy} \\
\Delta \textit{ClipboardPolicy} \\
\textit{path}? : \mathrm{seq}\, \mathbb{N}_1 \\
\hline
\textit{ooCache}' = \{p : \mathrm{seq}\, \mathbb{N}_1;\ u : \textit{UserID} \mid \\
\quad \textit{path}?\ \textit{prefix}\ p \land \textit{Node}(p) \mapsto u \in \textit{ObjectOwner} \bullet \\
\quad \textit{Node}((\#\textit{path}? + 1 \mathbin{..} \#p) \upharpoonright p) \mapsto u\} \cup \{p : \mathrm{seq}\, \mathbb{N}_1;\ n : \textit{Name};\ u : \textit{UserID} \mid \\
\quad \textit{path}?\ \textit{prefix}\ p \land \textit{Attribute}(p, n) \mapsto u \in \textit{ObjectOwner} \bullet \\
\quad \textit{Attribute}((\#\textit{path}? + 1 \mathbin{..} \#p) \upharpoonright p, n) \mapsto u\} \\
\textit{rpCache}' = \{r : \textit{RoleID};\ p : \mathrm{seq}\, \mathbb{N}_1;\ a : \textit{Action};\ c : \textit{Condition};\ ps : \mathbb{P}\, \textit{Provision} \mid \\
\quad \textit{path}?\ \textit{prefix}\ p \land r \mapsto (\textit{Node}(p), a, c, ps) \in \textit{RolePermission} \bullet \\
\quad r \mapsto (\textit{Node}((\#\textit{path}? + 1 \mathbin{..} \#p) \upharpoonright p), a, c, ps)\} \cup \\
\quad \{r : \textit{RoleID};\ p : \mathrm{seq}\, \mathbb{N}_1;\ n : \textit{Name};\ a : \textit{Action};\ c : \textit{Condition};\ ps : \mathbb{P}\, \textit{Provision} \mid \\
\quad \textit{path}?\ \textit{prefix}\ p \land r \mapsto (\textit{Attribute}(p, n), a, c, ps) \in \textit{RolePermission} \bullet \\
\quad r \mapsto (\textit{Attribute}((\#\textit{path}? + 1 \mathbin{..} \#p) \upharpoonright p, n), a, c, ps)\}
\end{array}
$$

Cutting a node means copying the node followed by deleting it, both when the content-related operation is considered (cf. *CutNodeC* in §3.1.3) and the policy-related one.

$$\textit{CutNodeP} \mathrel{\widehat{=}} \textit{CopyNodeP} \mathbin{;} \textit{DeleteNodeP}$$

With respect to the content, pasting a node basically means adding the node cached in the clipboard (via the *AddNodeC* operation). With respect to the policy, it is not; ownership information and permissions are not generated but pasted as well. To reduce the complexity of the *PasteNodeP* schema, we have specified two auxiliary functions:

Given an object to user mapping and the address of where the node is to be added, the following function returns the object to user mapping that provides the same ownership information in the tree with the added node.

$$
\begin{array}{l}
\textit{ShiftAddOOsF} : \mathbb{P}(\textit{Object} \times \textit{UserID}) \times \mathrm{seq}_1\, \mathbb{N}_1 \to \mathbb{P}(\textit{Object} \times \textit{UserID}) \\
\hline
\forall\, oo : \mathbb{P}(\textit{Object} \times \textit{UserID});\ p_1 : \mathrm{seq}_1\, \mathbb{N}_1 \bullet \\
\quad \textit{ShiftAddOOsF}(oo, p_1) = \{p : \mathrm{seq}\, \mathbb{N}_1;\ u : \textit{UserID} \mid \\
\quad \textit{Node}(p) \mapsto u \in oo \land \neg\, (p_1\ \textit{prefix}\ p) \bullet \\
\quad \textit{ShiftAddObjectF}(\textit{Node}(p), p_1) \mapsto u\} \cup \{p : \mathrm{seq}\, \mathbb{N}_1;\ n : \textit{Name};\ u : \textit{UserID} \mid \\
\quad \textit{Attribute}(p, n) \mapsto u \in oo \land \neg\, (p_1\ \textit{prefix}\ p) \bullet \\
\quad \textit{ShiftAddObjectF}(\textit{Attribute}(p, n), p_1) \mapsto u\}
\end{array}
$$

Given a role-permission relation and the address of where the node is to be added, the following function returns the role-permission relation in the tree with the added node.

$$
\begin{array}{l}
ShiftAddRPsF : \mathbb{P}(RoleID \times Permission) \times \mathrm{seq}_1\,\mathbb{N}_1 \rightarrow \mathbb{P}(RoleID \times Permission) \\
\hline
\forall\, rp : \mathbb{P}(RoleID \times Permission);\ p_1 : \mathrm{seq}_1\,\mathbb{N}_1 \bullet \\
\quad ShiftAddRPsF(rp, p_1) = \{r : RoleID;\ p : \mathrm{seq}\,\mathbb{N}_1;\ a : Action; \\
\quad c : Condition;\ ps : \mathbb{P}\,Provision \mid \\
\quad r \mapsto (Node(p), a, c, ps) \in rp \land \neg\,(p_1\ prefix\ p) \bullet \\
\quad r \mapsto (ShiftAddObjectF(Node(p), p_1), a, c, ps)\} \cup \{r : RoleID;\ p : \mathrm{seq}\,\mathbb{N}_1; \\
\quad n : Name;\ a : Action;\ c : Condition;\ ps : \mathbb{P}\,Provision \mid \\
\quad r \mapsto (Attribute(p, n), a, c, ps) \in rp \land \neg\,(p_1\ prefix\ p) \bullet \\
\quad r \mapsto (ShiftAddObjectF(Attribute(p, n), p_1), a, c, ps)\}
\end{array}
$$

$$
\begin{array}{l}
\textit{PasteNodeP} \\
\hline
\Delta DocumentPolicy \\
\Xi ClipboardPolicy \\
path? : \mathrm{seq}\,\mathbb{N}_1 \\
\hline
ObjectOwner' = ShiftAddOOsF(ObjectOwner, path?) \cup \\
\quad \{p : \mathrm{seq}\,\mathbb{N}_1;\ u : UserID \mid Node(p) \mapsto u \in ooCache \bullet \\
\quad Node(path? \frown p) \mapsto u\} \cup \{p : \mathrm{seq}\,\mathbb{N}_1;\ n : Name;\ u : UserID \mid \\
\quad Attribute(p, n) \mapsto u \in ooCache \bullet Attribute(path? \frown p, n) \mapsto u\} \\
RolePermission' = ShiftAddRPsF(RolePermission, path?) \cup \\
\quad \{r : RoleID;\ p : \mathrm{seq}\,\mathbb{N}_1;\ a : Action;\ c : Condition;\ ps : \mathbb{P}\,Provision \mid \\
\quad r \mapsto (Node(p), a, c, ps) \in rpCache \bullet r \mapsto (Node(path? \frown p), a, c, ps)\} \cup \\
\quad \{r : RoleID;\ p : \mathrm{seq}\,\mathbb{N}_1;\ n : Name;\ a : Action;\ c : Condition;\ ps : \mathbb{P}\,Provision \mid \\
\quad r \mapsto (Attribute(p, n), a, c, ps) \in rpCache \bullet \\
\quad r \mapsto (Attribute(path? \frown p, n), a, c, ps)\} \\
\hline
\end{array}
$$

Now we can formally express that reading, adding, deleting, copying, cutting, and pasting a node are transactions operating simultaneously on both the content and policy components.

$$
\begin{array}{l}
ReadNode \mathrel{\widehat=} ReadNodeC \land ReadNodeP \\
AddNode \mathrel{\widehat=} AddNodeC \land AddNodeP \\
DeleteNode \mathrel{\widehat=} DeleteNodeC \land DeleteNodeP \\
CopyNode \mathrel{\widehat=} CopyNodeC \land CopyNodeP \\
CutNode \mathrel{\widehat=} CutNodeC \land CutNodeP \\
PasteNode \mathrel{\widehat=} PasteNodeC \land PasteNodeP
\end{array}
$$

4.2.4 Operations on Attributes

Reading an attribute has no effect on the document's policy.

$$
\begin{array}{l}
\textit{ReadAttributeP} \\
\hline
\Xi DocumentPolicy \\
\hline
\end{array}
$$

Adding an attribute results in the user adding the attribute becoming the owner of the attribute.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddAttributeP}} \\
\hline
\Delta DocumentPolicy \\
subject? : Subject \\
path? : \mathrm{seq}\, \mathbb{N}_1 \\
name? : Name \\
\hline
ObjectOwner' = ObjectOwner \cup \{Attribute(path?, name?) \mapsto second(subject?)\} \\
RolePermission' = RolePermission \\
\hline
\end{array}
$$

Deleting an attribute removes all ownership information and all permissions related to the attribute.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DeleteAttributeP}} \\
\hline
\Delta DocumentPolicy \\
path? : \mathrm{seq}\, \mathbb{N}_1 \\
name? : Name \\
\hline
ObjectOwner' = ObjectOwner \setminus \{u : UserID \bullet Attribute(path?, name?) \mapsto u\} \\
RolePermission' = RolePermission \setminus \{rp : RolePermission \mid \\
\quad (\exists\, r : RoleID;\ a : Action;\ c : Condition;\ ps : \mathbb{P}\, Provision \bullet \\
\quad rp = r \mapsto (Attribute(path?, name?), a, c, ps))\} \\
\hline
\end{array}
$$

Changing an attribute (i.e., its value) has no effect on the document's policy. The owner is still the user who added the attribute, and the user who made the change does not gain any additional permissions.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ChangeAttributeP}} \\
\hline
\Xi DocumentPolicy \\
\hline
\end{array}
$$

Note that *ChangeAttributeP* cannot be defined analogously to *ChangeAttributeC* (i.e., not as $DeleteAttributeP \fatsemi AddAttributeP$).

Copying an attribute changes the clipboard policy but not the document policy. As specified in *CopyAttributeC* on page 10, the attribute being copied becomes the only attribute in the clipboard content's root node. Therefore, the path ($path?$) is removed in the course of copying those parts of the document policy related to the attribute being copied.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CopyAttributeP}} \\
\hline
\Xi DocumentPolicy \\
\Delta ClipboardPolicy \\
path? : \mathrm{seq}\, \mathbb{N}_1 \\
name? : Name \\
\hline
ooCache' = \{Attribute(\langle\rangle, name?) \mapsto ObjectOwner(Attribute(path?, name?))\} \\
rpCache' = \{r : RoleID;\ a : Action;\ c : Condition;\ ps : \mathbb{P}\, Provision \mid \\
\quad r \mapsto (Attribute(path?, name?), a, c, ps) \in RolePermission \bullet \\
\quad r \mapsto (Attribute(\langle\rangle, name?), a, c, ps)\} \\
\hline
\end{array}
$$

Cutting an attribute means copying the attribute followed by deleting it, whether the content-related operation is considered (cf. *CutAttributeC* in §3.1.3)

or the policy-related one.

$$CutAttributeP \mathrel{\widehat=} CopyAttributeP \mathbin{;} DeleteAttributeP$$

With respect to the content, pasting an attribute basically means adding the attribute cached in the clipboard (via the *AddAttributeC* operation). With respect to the policy, it is not; ownership information and permissions are not generated but pasted as well.

PasteAttributeP

$$\Delta DocumentPolicy$$
$$\Xi ClipboardPolicy$$
$$path? : \operatorname{seq} \mathbb{N}_1$$
$$name? : Name$$

$$ObjectOwner' = ObjectOwner \cup \{Attribute(path?, name?) \mapsto ooCache(Attribute(\langle\rangle, name?))\}$$
$$RolePermission' = RolePermission \cup \{r : RoleID;\ a : Action;\ c : Condition;\ ps : \mathbb{P}\, Provision \mid r \mapsto (Attribute(\langle\rangle, name?), a, c, ps) \in rpCache \bullet r \mapsto (Attribute(path?, name?), a, c, ps)\}$$

Now we can formally express that reading, adding, deleting, changing, copying, cutting, and pasting an attribute are transactions operating simultaneously on both the content and policy components.

$$ReadAttribute \mathrel{\widehat=} ReadAttributeC \land ReadAttributeP$$
$$AddAttribute \mathrel{\widehat=} AddAttributeC \land AddAttributeP$$
$$DeleteAttribute \mathrel{\widehat=} DeleteAttributeC \land DeleteAttributeP$$
$$ChangeAttribute \mathrel{\widehat=} ChangAttributeC \land ChangeAttributeP$$
$$CopyAttribute \mathrel{\widehat=} CopyAttributeC \land CopyAttributeP$$
$$CutAttribute \mathrel{\widehat=} CutAttributeC \land CutAttributeP$$
$$PasteAttribute \mathrel{\widehat=} PasteAttributeC \land PasteAttributeP$$

4.2.5 Policy-specific Operations

So far, the policy-related operations specify side effects that operations primarily related to the content have on the policy. When users perform these operations, they do so with the content in mind. In contrast, the next three operations are policy-specific and have no effect on the content whatsoever. They are adding a role-permission mapping, deleting a role-permission mapping, and delegating a *read*- or *change*-based permission.

Adding a role-permission mapping does not change any ownership information or existing role-permission mappings. The preconditions are that the policy invariants will be maintained (cf. *DocumentPolicy* on page 16).

$$
\begin{array}{l}
\textit{AddRolePermission} \\
\hline
\Delta \textit{DocumentPolicy} \\
\textit{role}? : \textit{RoleID} \\
\textit{permission}? : \textit{Permission} \\
\hline
\textit{permission}?.2 \neq \textit{change} \lor (\exists \textit{path} : \operatorname{seq} \mathbb{N}_1 \bullet \textit{permission}?.1 = \textit{Node}(\textit{path})) \\
\forall r : \textit{RoleID};\ p : \textit{Permission} \mid r \mapsto p \in \textit{RolePermission} \bullet r \neq \textit{role}? \lor \\
\quad p.1 \neq \textit{permission}?.1 \lor p.2 \neq \textit{permission}?.2 \lor \\
\quad \textit{first}(p.3) \cap \textit{first}(\textit{permission}?.3) = \varnothing \lor \\
\quad \textit{second}(p.3) \cap \textit{second}(\textit{permission}?.3) = \varnothing \\
\textit{ObjectOwner}' = \textit{ObjectOwner} \\
\textit{RolePermission}' = \textit{RolePermission} \cup \{\textit{role}? \mapsto \textit{permission}?\} \\
\hline
\end{array}
$$

Deleting a role-permission mapping does not change any ownership information either. The precondition is simply that the role-permission mapping to be deleted exists in the first place.

$$
\begin{array}{l}
\textit{DeleteRolePermission} \\
\hline
\Delta \textit{DocumentPolicy} \\
\textit{role}? : \textit{RoleID} \\
\textit{permission}? : \textit{Permission} \\
\hline
\textit{role}? \mapsto \textit{permission}? \in \textit{RolePermission} \\
\textit{ObjectOwner}' = \textit{ObjectOwner} \\
\textit{RolePermission}' = \textit{RolePermission} \setminus \{\textit{role}? \mapsto \textit{permission}?\} \\
\hline
\end{array}
$$

As far as updating the policy goes, delegating a permission to a role is equivalent to adding a new role-permission mapping. As we shall see below, these two operations differ in when they are granted.

$$\textit{DelegateEditPermission} \mathrel{\widehat{=}} \textit{AddRolePermission}$$

4.3 Policy Interpretation

In classical access-control architectures, the PDP interprets the policy to decide whether access is granted and the PEP enforces this decision. As explained in Section 4.1.2, in architectures with support for provisions, the control flow is more complex and additionally involves interaction with a provisions service. We have specified the PDP and provisions service in Z: the PDP takes a request and a policy and responds with grant or deny, and the provisions service takes a set of provisions and responds with an empty set (denoting that all provisions have been satisfied) or with the set of provisions not yet satisfied.

4.3.1 Provisions Service

The provision service's state consists of a relation that states which user has signed which agreement (and which agreement has been signed by which user) and a sequence of log entries that states which user's access has caused which message to be logged.

$$
\begin{array}{l}
\textit{ProvisionsService} \\
\hline
UserAgreement : UserID \leftrightarrow AgreementID \\
Logs : \mathrm{seq}(UserID \times Message)
\end{array}
$$

Checking which provisions have already been made results in *log* provisions being made at the time of check. To reduce the complexity of the MadeProvisions schema, we have specified the following auxiliary function that turns sets into sequences with the same elements:

$$
\begin{array}{l}
[X] \\
\hline
set2seq : \mathbb{P}\, X \rightarrow \mathrm{seq}\, X \\
\hline
set2seq(\{\}) = \langle\rangle \\
\forall\, x : X;\ xs : \mathbb{P}\, X \bullet set2seq(\{x\} \cup xs) = \langle x \rangle \frown set2seq(xs)
\end{array}
$$

$$
\begin{array}{l}
\textit{MadeProvisions} \\
\hline
\Delta ProvisionsService \\
user? : UserID \\
ps? : \mathbb{P}\, Provision \\
ps! : \mathbb{P}\, Provision \\
\hline
ps! = \{a : AgreementID \mid Sign(a) \in ps? \land user? \mapsto a \notin UserAgreement \bullet \\
\qquad Sign(a)\} \\
Logs' = Logs \frown set2seq(\{m : Message \mid Log(m) \in ps? \bullet (user?, Log(m))\})
\end{array}
$$

4.3.2 Policy Decision Point

The following characteristic set defines the predicate that expresses whether a given policy permits a given request:

$$
\begin{array}{l}
_\ permits\ _ : (RoleID \leftrightarrow Permission) \leftrightarrow Request \\
\hline
\forall\, rps : RoleID \leftrightarrow Permission;\ req : Request \bullet \\
\qquad rps\ permits\ req \Leftrightarrow (\exists_1\, p : Permission \bullet first(req.1) \mapsto p \in rps \land \\
\qquad req.2 = p.1 \land req.3 = p.2 \land req.4 \in first(p.3) \land \\
\qquad second(req.1) \in second(p.3))
\end{array}
$$

That is, a policy permits a request if and only if the role of the request subject maps to exactly one permission in the policy such that the request object and the permission object are equal, that the request action and the permission action are equal, that the request environment is in the set of permission environments, and that the request user is in the set of permission users. Note that to support hierarchical role-based access control, we not only have to change this predicate, but also the *AddRolePermission* schema and *DocumentPolicy*'s invariant as well as the auxiliary function *GetProvisionSet* below.

Given a request and a policy, *GetProvisionSet* returns the set of provisions that must have been made for the request to be granted. The specification of *GetProvisionSet* relies on an auxiliary function, *arb*, that given a set with exactly one element returns that element:

$$[X]$$
$$arb : \mathbb{P}_1 X \rightarrowtail X$$
$$\forall x : X \bullet arb(\{x\}) = x$$

$$GetProvisionSet : Request \times (RoleID \leftrightarrow Permission) \rightarrowtail \mathbb{P}\, Provision$$
$$\forall req : Request;\ rps : RoleID \leftrightarrow Permission \bullet GetProvisionSet(req, rps) = arb(\{p : Permission \mid first(req.1) \mapsto p \in rps \wedge req.2 = p.1 \wedge req.3 = p.2 \wedge req.4 \in first(p.3) \wedge second(req.1) \in second(p.3) \bullet p.4\})$$

The PDP evaluates requests while ignoring the hierarchical nature of the content. It is up to the PEP whether access to a node depends on access to all its ancestors or not. Except for the *ChangeNodeRequest* operation, which is internal to the PDP, the following request operations define the interface between PEP and PDP.

A read node request is (tentatively) granted if the subject requesting read access is the owner of the node or if the policy permits read access to the node.

ReadNodeRequest

$$\Xi DocumentPolicy$$
$$subject? : Subject$$
$$path? : \operatorname{seq} \mathbb{N}_1$$
$$environment? : Environment$$
$$response! : Response$$
$$response! = \textbf{if } ObjectOwner(Node(path?)) = second(subject?) \textbf{ then } Grant(\{\}) \textbf{ else if } RolePermission\ permits\ (subject?, Node(path?), read, environment?) \textbf{ then } Grant(GetProvisionSet((subject?, Node(path?), read, environment?), RolePermission)) \textbf{ else } Deny$$

A change node request is (tentatively) granted if the subject requesting change access is the owner of the node or if the policy permits *both* read and change access to the node. The provisions that must have been made is the union of those that must have been made for a read access and of those that must have been made for a change access.

ChangeNodeRequest

$$\Xi DocumentPolicy$$
$$subject? : Subject$$
$$path? : \operatorname{seq} \mathbb{N}_1$$
$$environment? : Envrionment$$
$$response! : Response$$
$$response! = \textbf{if } ObjectOwner(Node(path?)) = second(subject?) \textbf{ then } Grant(\{\}) \textbf{ else if } RolePermission\ permits\ (subject?, Node(path?), read, environment?) \wedge RolePermission\ permits\ (subject?, Node(path?), change, environment?) \textbf{ then } Grant(GetProvisionSet((subject?, Node(path?), read, environment?), RolePermission) \cup GetProvisionSet((subject?, Node(path?), change, environment?), RolePermission)) \textbf{ else } Deny$$

As explained in Section 4, add access to a node (here, adding a node to it) and delete access to a node (here, deleting a node from it) requires change access to it.

$$AddNodeRequest \mathrel{\widehat=} ChangeNodeRequest[front(path?)/path?]$$

$$DeleteNodeRequest \mathrel{\widehat=} ChangeNodeRequest[front(path?)/path?]$$

Copy access to a node requires read access to it.

$$CopyNodeRequest \mathrel{\widehat=} ReadNodeRequest$$

Cut access to a node requires both copy and delete access to it.

CutNodeRequest

$$\begin{array}{l}
\Xi DocumentPolicy \\
subject? : Subject \\
path? : \mathrm{seq}_1\, \mathbb{N}_1 \\
environment? : Environment \\
response! : Response \\
\hline
(\exists\, ps_1, ps_2 : \mathbb{P}\, Provision \bullet CopyNodeRequest[Grant(ps_1)/response!] \land \\
\quad DeleteNodeRequest[Grant(ps_2)/response!] \land \\
\quad response! = Grant(ps_1 \cup ps_2)) \lor \\
\quad ((CopyNodeRequest[Deny/response!] \lor \\
\quad DeleteNodeRequest[Deny/response!]) \land response! = Deny)
\end{array}$$

A node can be pasted where it can be added.

$$PasteNodeRequest \mathrel{\widehat=} AddNodeRequest$$

A read attribute request is (tentatively) granted if the subject requesting read access is the owner of the attribute or if the policy permits read access to the attribute.

ReadAttributeRequest

$$\begin{array}{l}
\Xi DocumentPolicy \\
subject? : Subject \\
path? : \mathrm{seq}\, \mathbb{N}_1 \\
name? : Name \\
environment? : Environment \\
response! : Response \\
\hline
response! = \mathbf{if}\ ObjectOwner(Attribute(path?, name?)) = second(subject?)\ \mathbf{then} \\
\quad Grant(\{\})\ \mathbf{else\ if}\ RolePermission\ permits\ (subject?, Attribute(path?, name?), \\
\quad read, environment?)\ \mathbf{then}\ Grant(GetProvisionSet((subject?, \\
\quad Attribute(path?, name?), read, environment?), RolePermission))\ \mathbf{else}\ Deny
\end{array}$$

As explained in Section 4, add access to a node (here, adding an attribute

to it) and delete access to a node (here, deleting an attribute from it) requires change access to it.

$AddAttributeRequest \mathrel{\widehat=} ChangeNodeRequest$

$DeleteAttributeRequest \mathrel{\widehat=} ChangeNodeRequest$

Change access to an attribute requires both read access to it and change access to its node.

ChangeAttributeRequest

$$
\begin{array}{l}
\Xi DocumentPolicy \\
subject? : Subject \\
path? : \mathrm{seq}\,\mathbb{N}_1 \\
name? : Name \\
environment? : Environment \\
response! : Response \\
\hline
(\exists ps_1, ps_2 : \mathbb{P}\,Provision \bullet ReadAttributeRequest[Grant(ps_1)/response!] \land \\
\quad ChangeNodeRequest[Grant(ps_2)/response!] \land \\
\quad response! = Grant(ps_1 \cup ps_2)) \lor \\
\quad ((ReadAttributeRequest[Deny/response!] \lor \\
\quad ChangeNodeRequest[Deny/response!]) \land response! = Deny)
\end{array}
$$

Copy access to an attribute requires read access to it.

$CopyAttributeRequest \mathrel{\widehat=} ReadAttributeRequest$

Cut access to an attribute requires both copy and delete access to it.

CutAttributeRequest

$$
\begin{array}{l}
\Xi DocumentPolicy \\
subject? : Subject \\
path? : \mathrm{seq}\,\mathbb{N}_1 \\
name? : Name \\
environment? : Environment \\
response! : Response \\
\hline
(\exists ps_1, ps_2 : \mathbb{P}\,Provision \bullet CopyAttributeRequest[Grant(ps_1)/response!] \land \\
\quad DeleteAttributeRequest[Grant(ps_2)/response!] \land \\
\quad response! = Grant(ps_1 \cup ps_2)) \lor \\
\quad ((CopyAttributeRequest[Deny/response!] \lor \\
\quad DeleteAttributeRequest[Deny/response!]) \land response! = Deny)
\end{array}
$$

An attribute can be pasted where it can be added.

$PasteAttributeRequest \mathrel{\widehat=} AddAttributeRequest$

Adding role-permission mappings can only be added and deleted by the

respective owner.

AddRolePermissionRequest

$\Xi DocumentPolicy$
$subject? : Subject$
$permission? : Permission$
$response! : Response$

$response! = \textbf{if}\ ObjectOwner(permission?.1) = second(subject?)\ \mathbf{then}$
$\quad Grant(\{\})\ \textbf{else}\ Deny$

DeleteRolePermissionRequest

$\Xi DocumentPolicy$
$subject? : Subject$
$permission? : Permission$
$response! : Response$

$response! = \textbf{if}\ ObjectOwner(permission?.1) = second(subject?)\ \mathbf{then}$
$\quad Grant(\{\})\ \textbf{else}\ Deny$

Delegating a permission requires delegate access on the respective object, that the action is *read* or *change*, that the condition of the new permission is never true when the condition of the original permission would not be, and that the original provisions must also be made in the new permission.

DelegateEditPermissionRequest

$\Xi DocumentPolicy$
$subject? : Subject$
$permission? : Permission$
$environment? : Environment$
$response! : Response$

$$response! = \textbf{if}\ RolePermission\ permits\ (subject?, permission?.1, delegate, environment?) \land permission?.2 \in \{read, change\} \land (\forall e : Environment \mid e \in first(permission?.3) \bullet RolePermission\ permits\ (subject?, permission?.1, permission?.2, e)) \land GetProvisionSet((subject?, permission?.1, permission?.2, e), RolePermission) \subseteq permission?.4\ \mathbf{then}\ Grant(GetProvisionSet((subject?, permission?.1, delegate, environment?), RolePermission))\ \textbf{else}\ Deny$$

4.3.3 Policy Enforcement Point

The modeling requirements for the PEP are different than for the other subsystems. The UI is event-driven and the PEP must synchronize (interact) with the UI as well as the other architectural components and its control flow is data-dependent. While Z is well-suited for data modeling, it cannot easily describe such process interaction. Therefore we employed CSP-OZ, which combines Z with the process calculus CSP as mentioned in the introduction.

A CSP-OZ class describes both operations (in Z) and their synchronization

(in CSP). The excerpt in Figure 5 formalizes the generic description given in Section 4.1.2 for the operation of reading a node. It leaves open the application-specific mechanisms of receiving events (e.g., *ReadNode_event*) and of updating the UI (e.g., *ReadNode_ret*). The other operations (copying a node, reading an attribute, etc.) are declared analogously. The formalization of the UI (cf. Figure 4) is similar but simpler than that of the PEP. CSP processes synchronize along so-called channels. Explicitly declared (`chan`) are the channels *login*, *logout*, and *abort*. Operations specified in Z are also channels. The UI and the PEP execute in parallel. At first, both processes are in the main loop (`main`) until a user successfully logs in, at which point they enter their event-processing loops (*PEPL* in the case of the PEP). When a user logs out via the UI, the PEP follows suit and returns to its main loop. Unlike the UI, the PEP must take additional steps between receiving an event and handling it or aborting, in order to make an access decision. First, it determines the current environment (*GetEnv*) and then it communicates with the PDP (via *ReadNodeRequest*) and, provided the PDP has not denied the request, with the provisions service (via *MadeProvisions*). The PEP signals access denied on the *abort* channel, which forces the UI to abort without having handled the event.

$| \; GetEnv : Environment$

UI

$$
\begin{array}{l}
\texttt{chan}\ login : [\ u? : UserID;\ ok! : \mathbb{B}\ ] \\
\texttt{chan}\ logout \\
\texttt{chan}\ abort \\[4pt]
\texttt{main} = login?u?ok \rightarrow \\
\qquad\quad (ok\ \&\ getSubject?subject \rightarrow UIL \\
\qquad\quad \Box\ \neg ok\ \&\ \texttt{main}) \\
UIL = logout \rightarrow \texttt{main} \\
\qquad\quad \Box\ ReadNode_event?path \rightarrow \\
\qquad\quad\quad ReadNode!path?attributesDom?childrenNr \rightarrow \\
\qquad\quad\quad ReadNode_ret!attributesDom!childrenNr \rightarrow UIL \\
\qquad\quad \Box\ abort \rightarrow UIL \\
\qquad\quad \Box\ \ldots
\end{array}
$$

Figure 4: User Interface

Hence the PEP class brings together the various Z specifications from before and formalizes how policies are interpreted and enforced. Overall, our model provides a precise description, with a formal mathematical semantics, of secure document processing, i.e., documents, operations on them, and access control.

4.4 Print Operation

We have postponed the discussion of the print operation until now because the content-related part cannot really be specified independent of the policy-related part.

$$
\begin{array}{l}
\textit{PEP} \\
\hline
\texttt{chan}\ login : [\ u? : UserID;\ ok! : \mathbb{B}\ ] \\
\texttt{chan}\ logout \\
\texttt{chan}\ abort \\
\texttt{main} = login?u?ok \rightarrow \\
\qquad (ok\ \&\ getSubject?subject \rightarrow PEPL(subject) \\
\qquad \Box\ \neg ok\ \&\ \texttt{main}) \\
PEPL(subject) = \\
\quad logout \rightarrow \texttt{main} \\
\quad \Box\ ReadNode_event?path \rightarrow GetEnv?environment \rightarrow \\
\qquad ReadNodeRequest!subject!path!environment?response \rightarrow \\
\qquad (response = Deny\ \&\ abort \rightarrow PEPL(subject) \\
\qquad \Box\ response = Grant(ps)\ \& \\
\qquad\quad MadeProvisions!subject!ps?rem_ps \rightarrow \\
\qquad\quad (rem_ps = \varnothing\ \& \\
\qquad\qquad ReadNode!path?attributesDom?childrenNr \rightarrow \\
\qquad\qquad ReadNode_ret!attributesDom!childrenNr \rightarrow PEPL(subject) \\
\qquad\quad \Box\ rem_ps \neq \varnothing\ \&\ abort \rightarrow PEPL(subject))) \\
\quad \Box\ \ldots
\end{array}
$$

Figure 5: Policy Enforcement Point

Given a printer as specified in the first schema below, a print operation could be simply defined as in the second schema if security were not a concern:

$$
\begin{array}{l}
\textit{Printer} \\
\hline
output : Tree
\end{array}
\qquad
\begin{array}{l}
\textit{Print} \\
\hline
\Xi Document \\
\Delta Printer \\
\hline
output' = root
\end{array}
$$

Since security is a concern, one idea is to prune the document and print the pruned document along the following lines:

$$
\begin{array}{l}
\textit{PruneAndPrint} \\
\hline
\Xi Document \\
\Delta Printer \\
subject? : Subject \\
environment? : Environment \\
\hline
PrunedTreeDomain = \{p : TreeDomainF(root) \mid (\exists ps : \mathbb{P}\,Provision \bullet \\
\qquad PrintNodeRequest[p, Grant(ps)/path?, response!] \\
\qquad \land MadeProvision[ps, \{\}/ps?, ps!])\} \\
PrintableAttributes = \{p : PrunedTreeDomain;\ n : Name \mid (\exists ps : \mathbb{P}\,Provision \bullet \\
\qquad PrintAttributeRequest[p, n, Grant(ps)/path?, name?, response!] \\
\qquad \land MadeProvision[ps, \{\}/ps?, ps!]) \bullet (p, n)\} \\
\ldots
\end{array}
$$

However, the handling of provisions is a problem. Should all permissions whose provisions have not been made at the time of printing be ignored? Or

should the user be told which provisions she could make to print more of a document? The answer is obviously application-specific. Therefore, we limit ourselves to specifying how print requests must be evaluated, but leave it to applications to make a trade-off between pruning and interacting with the user.

A print node request is (tentatively) granted if the subject requesting print access is the owner of the node or if the policy permits *both* read and print access to the node. The provisions that must have been made is the union of those that must have been made for a read access and of those that must have been made for a print access.

PrintNodeRequest

$\Xi DocumentPolicy$
$subject? : Subject$
$path? : \operatorname{seq} \mathbb{N}_1$
$environment? : Envrionment$
$response! : Response$

$response! = \textbf{if } ObjectOwner(Node(path?)) = second(subject?) \textbf{ then}$
$\quad Grant(\{\}) \textbf{ else if } RolePermission\ permits\ (subject?, Node(path?), read,$
$\quad environment?) \land RolePermission\ permits\ (subject?, Node(path?), print,$
$\quad environment?) \textbf{ then } Grant(GetProvisionSet((subject?, Node(path?), read,$
$\quad environment?), RolePermission) \cup GetProvisionSet((subject?,$
$\quad Node(path?), print, environment?), RolePermission)) \textbf{ else } Deny$

Similarly, a print attribute request is (tentatively) granted if the subject requesting print access is the owner of the attribute or if the policy permits *both* read and print access to the attribute. The provisions that must have been made is the union of those that must have been made for a read access and of those that must have been made for a print access.

PrintAttributeRequest

$\Xi DocumentPolicy$
$subject? : Subject$
$path? : \operatorname{seq} \mathbb{N}_1$
$name? : Name$
$environment? : Envrionment$
$response! : Response$

$response! = \textbf{if } ObjectOwner(Attribute(path?, name?)) = second(subject?) \textbf{ then}$
$\quad Grant(\{\}) \textbf{ else if } RolePermission\ permits\ (subject?, Attribute(path?, name?),$
$\quad read, environment?) \land RolePermission\ permits\ (subject?, Attribute(path?,$
$\quad name?), print, environment?) \textbf{ then } Grant(GetProvisionSet((subject?,$
$\quad Attribute(path?, name?), read, environment?), RolePermission) \cup$
$\quad GetProvisionSet((subject?, Attribute(path?, name?), print, environment?),$
$\quad RolePermission)) \textbf{ else } Deny$

5 Related Work

A number of commercial document-processing systems offer security functionality, for example those of Adobe[7] and Microsoft[8]. In contrast to our work, these systems offer only coarse-grained protection. Moreover, once data is copied, it is at the user's discretion.

The work closest to ours is in *XML access control*, which is an active research area concerned with controlling access to constituent parts of XML documents (e.g., [7, 15, 19, 2, 1, 10, 9]). We shall first discuss the primary characteristics of this work, and then examine several prominent proposals.

XML access control models are fine-grained, although the different proposals differ in their granularity and the types of their constituent parts. Moreover, they differ in the operations offered and their semantics. Some XML access control systems only provide a read operation with no arguments and thus expect their users to request entire XML documents (typically, there is only one instance, namely the XML-encoded database), in which case they respond with a censored copy called the *view* (Gabillon [9] compares several view-generation strategies). The current proposals differ considerably in the policy languages they offer, e.g., their features and syntactic sugar. Although these additions are intended to ease a policy writer's life, they are also a double-edged sword: they not only make the policy language more complex, they also necessitate conflict-resolution strategies [7, 15, 19]. In contrast, we have focused on a simple, yet expressive, core language with a clean, formal semantics. All current proposals for XML access control are limited in that they leave data at the user's discretion once it is copied. In contrast, we have solved this problem by adapting the idea of sticky policies to our model.

The XML Access Control Language (XACL) [15, 19] has up to now been the only proposal for XML access control with concrete support for provisions. Our model differs from that of the XACL, in this respect, in that provisions never have to be made (not even by the system) when access is denied. More importantly, XACL policies may be ambiguous in terms of which provisions must hold for a given request. In our model, at most one permission matches a request, so there is no such ambiguity.

Bertino *et al.* [2, 1] have proposed an approach to XML access control consisting of two parts: an access-control system *Author-*χ, and a credentials and policy language χ*-Sec*. Their proposal goes beyond XML access control in that they actually consider semi-structured data encoded in XML documents. They allow arcs (i.e., references or hyperlinks) to be secured with what they call the *navigate* privilege (privileges are what we call actions). Our model does not encompass semi-structured data. However, arcs can be encoded as attributes (as is done in the XML), whereby read access can be interpreted as navigate access. This proposal has a rich language for expressing temporal conditions, based on periodic time expressions [21]; from our requirements analysis, these are not necessary in our context.

Gabillon *et al.* [10, 9] go beyond XML access control and consider tree-structured data. However, nodes in their trees have no properties other than child nodes. This has the unfortunate consequence that the policy language

[7] `http://www.adobe.com/security/`
[8] `http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt/`

must be adapted to every specialization. In contrast, we can refine our document-content model without changing our policy language. The work by Gabillon *et al.* [10, 9] is the only closely related work in which object ownership and policy editing are not foreign concepts, and permission delegation is supported as well. All other approaches assume that policies are schema-based, static, and provided by (not further specified) administrators.

6 Conclusion

We have presented a formal model of an access-control system for document security. This model reflects real-world requirements and provides a precise design for solving this problem in a general way. Hence it represents a large step towards a general-purpose document-security system.

As future work, we will take the remaining steps in building a prototype implementation. First, we shall define a concrete syntax for our policy language and implement a PDP that interprets this syntax and can evaluate requests. A likely candidate for the concrete syntax is an eXtensible Access Control Markup Language (XACML) [22] profile in which case our PDP could be based on an existing XACML PDP, such as the one from Sun Microsystems Laboratories[9]. Because the XACML lacks a formal semantics, an alternative is to directly implement the PDP as a refinement of our formal model. Second, we will employ cryptographic mechanisms to secure documents during storage and while in transit so that only trusted systems can access them. Third, as a proof of concept, we will implement an XML editor along the lines of the architecture in Section 4.1.2 on page 14. Finally, we plan to embed the XML editor in a trustworthy client environment, where master keys are secured in Trusted Platform Modules (TPMs) [27]. All of these steps are realistic and should contribute to a practical solution that represents a large advance in the way that documents, and more generally hierarchically structured content, are secured.

7 Acknowledgments

We would like to thank Achim Brucker, Prof. Dr. Ernst-Rüdiger Olderog, and Dr. Burkhart Wolff for their help with Z and Prof. Olderog also for his help with CSP; Dr. Günter Karjoth for his help with policy languages; and Beat Perjés and Dr. Gritta Wolf for their help with gathering requirements.

For their valuable feedback on earlier versions of this text, we would also like to thank Dr. Karjoth and Dr. Wolf as well as Manuel Hilty, Michael Näf, and Dr. Alexander Pretschner.

A Proof of Concept: XML Documents

In this section, we refine the model for abstract documents to a model for XML documents. As we shall see, it suffices to refine the content model; the policy model need not be refined in any manner!

[9] `http://sunxacml.sourceforge.net/`

A.1 Data Types

First we reserve two attribute names. One for the attribute whose value—or non-existence—determines the type of XML node. And one for the attribute whose value determines the actual value of a comment node, processing-instruction node, or text node.

$$\begin{array}{|l} \mathit{nameName}, \mathit{valueName} : \mathit{Name} \\ \hline \mathit{nameName} \neq \mathit{valueName} \end{array}$$

Then we reserve two attribute values. One for the name of comment nodes, which is "!--" in a standard XML encoding, and one for the name of processing-instruction nodes, which is "?". All other *nameName* attribute values denote tags in element nodes.

$$\begin{array}{|l} \mathit{commentNameValue}, \mathit{piNameValue} : \mathit{Value} \\ \hline \mathit{commentNameValue} \neq \mathit{piNameValue} \end{array}$$

Comments are nodes that have two attributes and no children. The two attributes are a *nameName* attribute with value *commentNameValue* and a *valueName* attribute whose value is the actual comment. *Processing instructions* are also nodes that have two attributes and no children. The two attributes are a *nameName* attribute with value *piNameValue* and a *valueName* attribute whose value is the actual processing instruction. *Elements* are nodes that have one or more attributes and zero or more child nodes. One attribute is the *nameName* attribute whose value is the element's tag and the other attributes are the actual XML attributes, but none of the attributes is a *valueName* attribute. *Texts* are nodes that have one attribute and no child node. The attribute is a *valueName* attribute whose value is the actual text.

$$\begin{array}{l}
\mathit{Comment} == \{t : \mathit{Tree} \mid (\exists\, v : \mathit{Value} \bullet \\
\quad t = \mathit{Node}(\{\mathit{nameName} \mapsto \mathit{commentNameValue}, \mathit{valueName} \mapsto v\}, \langle\rangle))\} \\
\mathit{PI} == \{t : \mathit{Tree} \mid (\exists\, v : \mathit{Value} \bullet \\
\quad t = \mathit{Node}(\{\mathit{nameName} \mapsto \mathit{piNameValue}, \mathit{valueName} \mapsto v\}, \langle\rangle))\} \\
\mathit{Element} == \{t : \mathit{Tree} \mid (\exists\, a : \mathit{Attributes};\ ts : \mathrm{seq}\, \mathit{Tree} \mid \\
\quad \mathit{nameName} \in \mathsf{dom}\, a \land \mathit{valueName} \notin \mathsf{dom}\, a \land \\
\quad (\mathit{nameName} \mapsto \mathit{commentNameValue}) \notin a \land (\mathit{nameName} \mapsto \mathit{piNameValue}) \notin a \bullet \\
\quad t = \mathit{Node}(a, ts))\} \\
\mathit{Text} == \{t : \mathit{Tree} \mid (\exists\, v : \mathit{Value} \bullet \\
\quad t = \mathit{Node}(\{\mathit{valueName} \mapsto v\}, \langle\rangle))\} \\
\mathit{XmlNode} == \mathit{PI} \cup \mathit{Element} \cup \mathit{Text}
\end{array}$$

While we defined the set of comments for the sake of completeness, we do not include it in the set of XML nodes hereafter. XML comments are first and foremost a means for users to comment XML documents that they edit directly and can be ignored by XML parsers. In our context, users are never allowed to directly access an XML document's encoding.

A.2 Containers (State)

Except for the document root, an XML document consists of processing-instruction nodes, element nodes, and text nodes only. At most one element node, namely the root element, is a child of the document root. No text node is the child of the document root. And text nodes are never adjacent (cf. *AddTextNodeC* operation schema below).

XmlDocument

$$
\begin{array}{l}
\mathit{DocumentContent} \\
\hline
\exists\, ts : \operatorname{seq} \mathit{Tree} \bullet \mathit{root} = \mathit{Node}(\{\}, ts) \\
\forall\, p : \mathit{TreeDomainF}(\mathit{root}) \mid p \neq \langle\rangle \bullet \mathit{ReadNodeF}(\mathit{root}, p) \in \mathit{XmlNode} \\
\forall\, p_1, p_2 : \mathit{TreeDomainF}(\mathit{root}) \mid \\
\quad \mathit{ReadNodeF}(\mathit{root}, p_1) \in \mathit{Element} \land \mathit{ReadNodeF}(\mathit{root}, p_2) \in \mathit{Element} \bullet \\
\quad p_1 = p_2 \lor \# p_1 > 1 \lor \# p_2 > 1 \\
\forall\, p : \mathit{TreeDomainF}(\mathit{root}) \mid \mathit{ReadNodeF}(\mathit{root}, p) \in \mathit{Text} \bullet \# p > 1 \\
\forall\, p_1, p_2 : \mathit{TreeDomainF}(\mathit{root}) \mid \\
\quad \mathit{ReadNodeF}(\mathit{root}, p_1) \in \mathit{Text} \land \mathit{ReadNodeF}(\mathit{root}, p_2) \in \mathit{Text} \bullet \\
\quad \mathit{front}(p_1) \neq \mathit{front}(p_2) \lor \mathit{AbsValF}(\mathit{last}(p_1) - \mathit{last}(p_2)) \neq 1
\end{array}
$$

A policed XML document is a pair consisting of a plain XML document and a document policy.

$$\mathit{PolicedXmlDocument} \mathrel{\widehat=} \mathit{XmlDocument} \land \mathit{DocumentPolicy}$$

A.3 Discovery Operations

The (abstract) *ReadNodeC* operation is wrapped by three discovery operations, one that returns a node's type, one that returns a document root's or element's number of children, and one that returns the domain of an element's attributes. Note that the policy-related component of the operations is the *ReadNodeP* operation and that the requests are equivalent to a read node request.

GetTypeC

$$
\begin{array}{l}
\Xi \mathit{XmlDocument} \\
\mathit{path}? : \operatorname{seq}_1 \mathbb{N}_1 \\
\mathit{type}! : \mathbb{P}\, \mathit{XmlNode} \\
\hline
(\exists\, \mathit{ad} : \mathbb{P}\, \mathit{Name} \mid \mathit{nameName} \in \mathit{ad} \land \mathit{valueName} \in \mathit{ad} \bullet \\
\quad \mathit{ReadNodeC}[\mathit{ad}/\mathit{attributesDom}!]) \land \mathit{type}! = \mathit{PI} \\
\quad \lor (\exists\, \mathit{ad} : \mathbb{P}\, \mathit{Name} \mid \mathit{nameName} \in \mathit{ad} \land \mathit{valueName} \notin \mathit{ad} \bullet \\
\quad \mathit{ReadNodeC}[\mathit{ad}/\mathit{attributesDom}!]) \land \mathit{type}! = \mathit{Element} \\
\quad \lor (\exists\, \mathit{ad} : \mathbb{P}\, \mathit{Name} \mid \mathit{nameName} \notin \mathit{ad} \land \mathit{valueName} \in \mathit{ad} \bullet \\
\quad \mathit{ReadNodeC}[\mathit{ad}/\mathit{attributesDom}!]) \land \mathit{type}! = \mathit{Text}
\end{array}
$$

$$\mathit{GetType} \mathrel{\widehat=} \mathit{GetTypeC} \land \mathit{ReadNodeP}$$
$$\mathit{GetTypeRequest} \mathrel{\widehat=} \mathit{ReadNodeRequest}$$

GetNrOfChildrenC

$$\Xi XmlDocument$$
$$path? : \text{seq}\, \mathbb{N}_1$$
$$childrenNr! : \mathbb{N}$$

$$path? = \langle\rangle \lor ReadNodeF(root, path?) \in Element$$
$$ReadNodeC$$

$$GetNrOfChildren \mathrel{\widehat=} GetNrOfChildrenC \land ReadNodeP$$
$$GetNrOfChildrenRequest \mathrel{\widehat=} ReadNodeRequest$$

GetAttributesDomC

$$\Xi XmlDocument$$
$$path? : \text{seq}_1\, \mathbb{N}_1$$
$$aDom! : \mathbb{P}\, Name$$

$$ReadNodeF(root, path?) \in Element$$
$$nameName \notin aDom!$$
$$ReadNodeC[aDom! \cup nameName / attributesDom!]$$

$$GetAttributesDom \mathrel{\widehat=} GetAttributesDomC \land GetAttributesDomP$$
$$GetAttributesDomRequest \mathrel{\widehat=} ReadNodeRequest$$

A.4 Operations on Elements

ReadElementNameC

$$\Xi XmlDocument$$
$$path? : \text{seq}_1\, \mathbb{N}_1$$
$$value! : Value$$

$$ReadNodeF(root, path?) \in Element$$
$$ReadAttributeC[nameName / name?]$$

$$ReadElementName \mathrel{\widehat=} ReadElementNameC \land ReadAttributeP[nameName / name?]$$
$$ReadElementNameRequest \mathrel{\widehat=} ReadAttributeRequest[nameName / name?]$$

AddElementNodeC

$$\Delta XmlDocument$$
$$path? : \text{seq}_1\, \mathbb{N}_1$$
$$value? : Value$$
$$attributes? : Attributes$$
$$treeSequence? : \text{seq}\, XmlNode$$

$$\#path? > 1 \lor (\forall p : TreeDomainF(tree) \bullet ReadNodeF(tree, p) \notin Element)$$
$$nameName \notin attributes? \land valueName \notin attributes?$$
$$AddNodeC[attributes? \cup \{nameName \mapsto value?\} / attributes?]$$

$AddElementNode \mathrel{\widehat=} AddElementNodeC \land$
$\quad AddNodeP[attributes? \cup \{nameName \mapsto value?\}/attributes?]$
$AddElementNodeRequest \mathrel{\widehat=} AddNodeRequest$

DeleteElementNodeC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\, \mathbb{N}_1$

$ReadNodeF(root, path?) \in Element$
$DeleteNodeC$

$DeleteElementNode \mathrel{\widehat=} DeleteElementNodeC \land DeleteNodeP$
$DeleteElementNodeRequest \mathrel{\widehat=} DeleteNodeRequest$

ChangeElementNameC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\, \mathbb{N}_1$
$value? : Value$

$ReadNodeF(root, path?) \in Element$
$ChangeAttributeC[nameName/name?]$

$ChangeElementName \mathrel{\widehat=} ChangeElementNameC \land ChangeAttributeP[nameName/name?]$
$ChangeElementNameRequest \mathrel{\widehat=} ChangeAttributeRequest[nameName/name?]$

CopyElementNameC

$\Xi XmlDocument$
$\Delta ClipboardContent$
$path? : \mathrm{seq}_1\, \mathbb{N}_1$

$ReadNodeF(root, path?) \in Element$
$CopyAttributeC[nameName/name?]$

$CopyElementName \mathrel{\widehat=} CopyElementNameC \land CopyAttributeP[nameName/name?]$
$CopyElementNameRequest \mathrel{\widehat=} CopyAttributeRequest[nameName/name?]$

CopyElementNodeC

$\Xi XmlDocument$
$\Delta ClipboardContent$
$path? : \mathrm{seq}_1\, \mathbb{N}_1$

$ReadNodeF(root, path?) \in Element$
$CopyNodeC$

$CopyElementNode \mathrel{\widehat=} CopyElementNodeC \land CopyNodeP$
$CopyElementNodeRequest \mathrel{\widehat=} CopyNodeRequest$

$$
\begin{array}{l}
\textit{CutElementNodeC} \\
\hline
\Delta XmlDocument \\
\Delta ClipboardContent \\
path? : \mathrm{seq}_1\, \mathbb{N}_1 \\
\hline
ReadNodeF(root, path?) \in Element \\
CutNodeC
\end{array}
$$

$CutElementNode \mathrel{\widehat=} CutElementNodeC \land CutNodeP$
$CutElementNodeRequest \mathrel{\widehat=} CutNodeRequest$

$$
\begin{array}{l}
\textit{PasteElementNodeC} \\
\hline
\Delta XmlDocument \\
\Xi ClipboardContent \\
path? : \mathrm{seq}_1\, \mathbb{N}_1 \\
\hline
cCache \in Element \\
PasteNodeC
\end{array}
$$

$PasteElementNode \mathrel{\widehat=} PasteElementNodeC \land PasteNodeP$
$PasteElementNodeRequest \mathrel{\widehat=} PasteNodeRequest$

A.5 Operations on Attributes

$$
\begin{array}{l}
\textit{ReadXmlAttributeC} \\
\hline
\Xi XmlDocument \\
path? : \mathrm{seq}_1\, \mathbb{N}_1 \\
name? : Name \setminus \{nameName, valueName\} \\
value! : Value \\
\hline
ReadNodeF(root, path?) \in Element \\
ReadAttributeC
\end{array}
$$

$ReadXmlAttribute \mathrel{\widehat=} ReadXmlAttributeC \land ReadAttributeP$
$ReadXmlAttributeRequest \mathrel{\widehat=} ReadAttributeRequest$

AddXmlAttributeC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$
$name? : Name \setminus \{nameName, valueName\}$
$value? : Value$

$ReadNodeF(root, path?) \in Element$
$AddAttributeC$

$AddXmlAttribute \mathrel{\widehat{=}} AddXmlAttributeC \land AddAttributeP$
$AddXmlAttributeRequest \mathrel{\widehat{=}} AddAttributeRequest$

DeleteXmlAttributeC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$
$name? : Name \setminus \{nameName, valueName\}$

$ReadNodeF(root, path?) \in Element$
$DeleteAttributeC$

$DeleteXmlAttribute \mathrel{\widehat{=}} DeleteXmlAttributeC \land DeleteAttributeP$
$DeleteXmlAttributeRequest \mathrel{\widehat{=}} DeleteAttributeRequest$

ChangeXmlAttributeC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$
$name? : Name \setminus \{nameName, valueName\}$
$value? : Value$

$ReadNodeF(root, path?) \in Element$
$ChangeAttributeC$

$ChangeXmlAttribute \mathrel{\widehat{=}} ChangeXmlAttributeC \land ChangeAttributeP$
$ChangeXmlAttributeRequest \mathrel{\widehat{=}} ChangeAttributeRequest$

CopyXmlAttributeC

$\Xi XmlDocument$
$\Delta ClipboardContent$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$
$name? : Name \setminus \{nameName, valueName\}$

$ReadNodeF(root, path?) \in Element$
$CopyAttributeC$

$CopyXmlAttribute \mathrel{\widehat=} CopyXmlAttributeC \land CopyAttributeP$
$CopyXmlAttributeRequest \mathrel{\widehat=} CopyAttributeRequest$

CutXmlAttributeC

$\Delta XmlDocument$
$\Delta ClipboardContent$
$path? : \mathrm{seq}_1 \mathbb{N}_1$
$name? : Name \setminus \{nameName, valueName\}$

$ReadNodeF(root, path?) \in Element$
$CutAttributeC$

$CutXmlAttribute \mathrel{\widehat=} CutXmlAttributeC \land CutAttributeP$
$CutXmlAttributeRequest \mathrel{\widehat=} CutAttributeRequest$

PasteXmlAttributeC

$\Delta XmlDocument$
$\Xi ClipboardContent$
$path? : \mathrm{seq}_1 \mathbb{N}_1$
$name? : Name \setminus \{nameName, valueName\}$

$ReadNodeF(root, path?) \in Element$
$PasteAttributeC$

$PasteXmlAttribute \mathrel{\widehat=} PasteXmlAttributeC \land PasteAttributeP$
$PasteXmlAttributeRequest \mathrel{\widehat=} PasteAttributeRequest$

A.6 Operations on Texts

ReadTextValueC

$\Xi XmlDocument$
$path? : \mathrm{seq}_1 \mathbb{N}_1$
$value! : Value$

$ReadNodeF(root, path?) \in Text$
$ReadAttributeC[valueName/name?]$

$ReadTextValue \mathrel{\widehat=} ReadTextValueC \land ReadAttributeP[valueName/name?]$
$ReadTextValueRequest \mathrel{\widehat=} ReadAttributeRequest[valueName/name?]$

According to the Document Object Model (DOM) specification [28] (which we abstract from), "[w]hen a document is first made available via the DOM, there is only one `Text` node for each block of text. Users may create adjacent `Text` nodes that represent the contents of a given element without any intervening markup, but should be aware that there is no way to represent the

separations between these nodes in XML [...], so they will not (in general) persist between DOM editing sessions. The `Node.normalize()` method merges any such adjacent `Text` objects into a single node for each block of text." As we anticipate a security policy and do not want to worry about what it would mean to normalize rules referring to adjacent text nodes, we go further than the DOM in that in our model subjects cannot create them in the first place. Adding text is adding a new text leaf:

AddTextNodeC

$$
\begin{array}{l}
\Delta XmlDocument \\
path? : \mathrm{seq}_1\, \mathbb{N}_1 \\
value? : Value \\
\hline
ReadNodeF(root, front(path?)) \in Element \\
\forall p : TreeDomainF(root) \mid ReadNodeF(root, p) \in Text \bullet \\
\quad front(p) \neq front(path?) \lor AbsValF(last(p) - last(path?)) > 1 \\
AddNodeC[\{valueName \mapsto value?\}, \langle\rangle / attributes?, treeSequence?]
\end{array}
$$

$$
\begin{array}{l}
AddTextNode \mathrel{\widehat=} AddTextNodeC \land \\
\quad AddNodeP[\{valueName \mapsto value?\}, \langle\rangle / attributes?, treeSequence?] \\
AddTextNodeRequest \mathrel{\widehat=} AddNodeRequest
\end{array}
$$

DeleteTextNodeC

$$
\begin{array}{l}
\Delta XmlDocument \\
path? : \mathrm{seq}_1\, \mathbb{N}_1 \\
\hline
ReadNodeF(root, path?) \in Text \\
DeleteNodeC
\end{array}
$$

$$
\begin{array}{l}
DeleteTextNode \mathrel{\widehat=} DeleteTextNodeC \land DeleteNodeP \\
DeleteTextNodeRequest \mathrel{\widehat=} DeleteNodeRequest
\end{array}
$$

ChangeTextValueC

$$
\begin{array}{l}
\Delta XmlDocument \\
path? : \mathrm{seq}_1\, \mathbb{N}_1 \\
value? : Value \\
\hline
ReadNodeF(root, path?) \in Text \\
ChangeAttributeC[valueName / name?]
\end{array}
$$

$$
\begin{array}{l}
ChangeTextValue \mathrel{\widehat=} ChangeTextValue \land ChangeAttributeP[valueName / name?] \\
ChangeTextValueRequest \mathrel{\widehat=} ChangeAttributeRequest
\end{array}
$$

$$
\begin{array}{|l}
\text{— } \mathit{CopyTextValueC} \text{ —} \\
\Xi \mathit{XmlDocument} \\
\Delta \mathit{ClipboardContent} \\
\mathit{path}? : \mathrm{seq}_1\, \mathbb{N}_1 \\
\hline
\mathit{ReadNodeF}(\mathit{root}, \mathit{path}?) \in \mathit{Text} \\
\mathit{CopyAttributeC}[\mathit{valueName}/\mathit{name}?] \\
\hline
\end{array}
$$

$\mathit{CopyTextValue} \mathrel{\widehat=} \mathit{CopyTextValueC} \land \mathit{CopyAttributeP}[\mathit{valueName}/\mathit{name}?]$
$\mathit{CopyTextValueRequest} \mathrel{\widehat=} \mathit{CopyAttributeRequest}[\mathit{valueName}/\mathit{name}?]$

$$
\begin{array}{|l}
\text{— } \mathit{CutTextNodeC} \text{ —} \\
\Delta \mathit{XmlDocument} \\
\Delta \mathit{ClipboardContent} \\
\mathit{path}? : \mathrm{seq}_1\, \mathbb{N}_1 \\
\hline
\mathit{ReadNodeF}(\mathit{content}, \mathit{path}?) \in \mathit{Text} \\
\mathit{CutNodeC} \\
\hline
\end{array}
$$

$\mathit{CutTextNode} \mathrel{\widehat=} \mathit{CutTextNodeC} \land \mathit{CutNodeP}$
$\mathit{CutTextNodeRequest} \mathrel{\widehat=} \mathit{CutNodeRequest}$

$$
\begin{array}{|l}
\text{— } \mathit{PasteTextNodeC} \text{ —} \\
\Delta \mathit{XmlDocument} \\
\Xi \mathit{ClipboardContent} \\
\mathit{path}? : \mathrm{seq}_1\, \mathbb{N}_1 \\
\hline
\mathit{cCache} \in \mathit{Text} \\
\mathit{PasteNodeC} \\
\hline
\end{array}
$$

$\mathit{PasteTextNode} \mathrel{\widehat=} \mathit{PasteTextNodeC} \land \mathit{PasteNodeP}$
$\mathit{PasteTextNodeRequest} \mathrel{\widehat=} \mathit{PasteNodeRequest}$

A.7 Operations on Processing Instructions

$$
\begin{array}{|l}
\text{— } \mathit{ReadPIValueC} \text{ —} \\
\Xi \mathit{XmlDocument} \\
\mathit{path}? : \mathrm{seq}_1\, \mathbb{N}_1 \\
\mathit{value}! : \mathit{Value} \\
\hline
\mathit{ReadNodeF}(\mathit{root}, \mathit{path}?) \in \mathit{PI} \\
\mathit{ReadAttributeC}[\mathit{valueName}/\mathit{name}?] \\
\hline
\end{array}
$$

$\mathit{ReadPIValue} \mathrel{\widehat=} \mathit{ReadPIValueC} \land \mathit{ReadAttributeP}[\mathit{valueName}/\mathit{name}?]$
$\mathit{ReadPIValueRequest} \mathrel{\widehat=} \mathit{ReadAttributeRequest}[\mathit{valueName}/\mathit{name}?]$

AddPINodeC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$
$value? : Value$

$ReadNodeF(root, front(path?)) \in Element$
$AddNodeC[\{nameName \mapsto piNameValue, valueName \mapsto value?\}, \langle\rangle / attributes?, treeSequence?]$

$AddPINode \mathrel{\widehat=} AddPINodeC \land AddNodeP[\{nameName \mapsto piNameValue, valueName \mapsto value?\}, \langle\rangle / attributes?, treeSequence?]$
$AddPINodeRequest \mathrel{\widehat=} AddNodeRequest$

DeletePINodeC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$

$ReadNodeF(root, path?) \in PI$
$DeleteNodeC$

$DeletePINode \mathrel{\widehat=} DeletePINodeC \land DeleteNodeP$
$DeletePINodeRequest \mathrel{\widehat=} DeleteNodeRequest$

ChangePIValueC

$\Delta XmlDocument$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$
$value? : Value$

$ReadNodeF(root, path?) \in PI$
$ChangeAttributeC[valueName/name?]$

$ChangePIValue \mathrel{\widehat=} ChangePIValue \land ChangeAttributeP[valueName/name?]$
$ChangePIValueRequest \mathrel{\widehat=} ChangeAttributeRequest[valueName/name?]$

CopyPIValueC

$\Xi XmlDocument$
$\Delta ClipboardContent$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$

$ReadNodeF(root, path?) \in PI$
$CopyAttributeC[valueName/name?]$

$CopyPIValue \mathrel{\widehat=} CopyPIValueC \land CopyAttributeP[valueName/name?]$
$CopyPIValueRequest \mathrel{\widehat=} CopyAttributeRequest[valueName/name?]$

CutPINodeC

$\Delta XmlDocument$
$\Delta ClipboardContent$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$

$ReadNodeF(content, path?) \in PI$
$CutNodeC$

$CutPINode \mathrel{\widehat=} CutPINodeC \wedge CutNodeP$
$CutPINodeRequest \mathrel{\widehat=} CutNodeRequest$

PastePINodeC

$\Delta XmlDocument$
$\Xi ClipboardContent$
$path? : \mathrm{seq}_1\ \mathbb{N}_1$

$cCache \in PI$
$PasteNodeC$

$PastePINode \mathrel{\widehat=} PastePINodeC \wedge PasteNodeP$
$PastePINodeRequest \mathrel{\widehat=} PasteNodeRequest$

A.8 Remark

Several document formats are based on the XML. Thus, such documents can be secured by securing their XML-based encodings. Users could define application-level policies whose objects are, for example, chapters, lists, etc., and the system would map them to XML-level policies as suggested in Figure 6. ("XML-level policy" is a bit of a misnomer, though, as there is no need to refine our policy model to govern access to XML documents.)

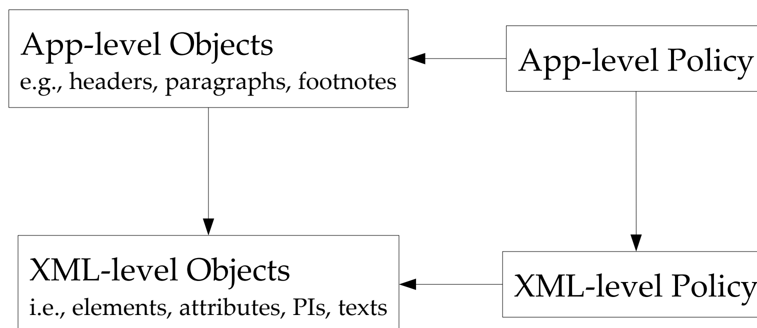

Figure 6: Mapping Application-level to XML-level Objects and Policies

Furthermore, if the application-level document model is a refinement of

our content model, the detour via the XML is not even necessary! Indeed, the application-specific document could be directly modeled along the lines of $AppSpecificDoc \mathrel{\widehat{=}} AppDocContent \land DocumentPolicy$.

References

[1] E. Bertino, B. Carminati, and E. Ferrari. "Access Control for XML documents and data." In *Information Security Technical Report*, vol. 9, no. 3, pp. 19–34, July-September 2004.

[2] E. Bertino, S. Castano, and E. Ferrari. "Securing XML Documents with Author-X." In *IEEE Internet Computing*, vol. 5, no. 3, pp. 21–31, May/June 2001.

[3] E. Bertino and R. Sandhu. "Database Security—Concepts, Approaches, and Challenges." In *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 2–19, January-March 2005.

[4] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. "Provisions and Obligations in Policy Rule Management." In *Journal of Network and Systems Management*, vol. 11, no. 3, pp. 351–372, September 2003.

[5] C. Bettini, S. Jajodia, X. S. Wang, and D. Wijesekera. "Reasoning with advanced policy rules and its application to access control." In *International Journal on Digital Libraries*, vol. 4, no. 3, pp. 156–170, November 2004.

[6] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. ACM Press, 1995. ISBN 0-201-59375-0.

[7] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. "A Fine-Grained Access Control System for XML Documents." In *ACM Transactions on Information and System Security*, vol. 5, no. 2, pp. 169–202, May 2002.

[8] C. Fischer. "CSP-OZ: a combination of Object-Z and CSP." In *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pp. 423–438. 1997.

[9] A. Gabillon. "An Authorization Model for XML DataBases." In *Proceedings of the 11th ACM conference on Computer and communications security*. 2004.

[10] A. Gabillon, M. Munier, J.-J. Bascou, L. Gallon, and E. Bruno. "An Access Control Model for Tree Data Structures." In *Proceedings of the 5th International Conference on Information Security*, pp. 117–135. 2002.

[11] S. Garfinkel, G. Spafford, and A. Schwartz. *Practical UNIX and Internet Security*. O'Reilly, 3rd ed., 2003. ISBN 0-596-00323-4.

[12] E. R. Harold and W. S. Means. *XML in a Nutshell*. O'Reilly, 3rd ed., 2004. ISBN 0-596-00764-7.

[13] M. Hilty, D. Basin, and A. Pretschner. "On Obligations." In S. de Capitani di Vimercati, P. Syverson, and D. Gollmann (eds.), *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, vol. 3679 of *Lecture Notes in Computer Science*, pp. 98–117. Springer-Verlag, September 2005.

[14] C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[15] IBM Tokyo Research Laboratory. "XML Access Control Language (XACL)." WWW.
URL `http://www.trl.ibm.com/projects/xml/xacl/`

[16] IBM Zurich Research Laboratory. "Enterprise Privacy Technologies." WWW.
URL `http://www.zurich.ibm.com/security/enterprise-privacy/`

[17] International Organization for Standardization. *Information technology – Z formal specification notation – Syntax, type system and semantics*, 1st ed., July 2002.
URL `http://www.iso.ch/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=21573`

[18] G. Karjoth, M. Schunter, and M. Waidner. "Platform for Enterprise Privacy Practices: Privacy-Enabled Management of Customer Data." In R. Dingledine and P. Syverson (eds.), *Privacy Enhancing Technologies*, vol. 2482 of *Lecture Notes in Computer Science*, pp. 69–84. Springer-Verlag, 2003.

[19] M. Kudo and S. Hada. "XML Document Security based on Provisional Authorization." In *Proceedings of the 7th ACM conference on Computer and communications security*, pp. 87–96. Athens, November 2000.

[20] A. Møller and M. Schwartzback. *An Introduction to XML and Web Technologies*. Addison Wesley Professional, 2006. ISBN 0-321-26966-7.

[21] M. Niézette and J.-M. Stévenne. "An Efficient Symbolic Representation of Periodic Time." In *Proceedings of the ISMM International Conference on Information and Knowledge Management (CIKM-92)*, pp. 161–168. 1992.

[22] OASIS. "eXtensible Access Control Markup Language (XACML)." Specification.
URL `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`

[23] J. Park and R. Sandhu. "The UCON_{ABC} Usage Control Model." In *ACM Transactions on Information and System Security*, vol. 7, no. 1, pp. 128–174, February 2004.

[24] P. E. Sevinç, D. Basin, and E.-R. Olderog. "Controlling Access to Documents: A Formal Access Control Model." In G. Müller (ed.), *Proceedings of the 1st International Conference on Emerging Trends in Information and Communication Security (ETRICS 2006)*, vol. 3995 of *Lecture Notes in Computer Science*, pp. 352–367. Springer-Verlag, June 2006.

[25] B. Smith and B. Komar. *Microsoft Windows Security Resource Kit*. Microsoft Press, 2nd ed., 2005. ISBN 0-735-62174-8.

[26] A. G. Stoica and C. Farkas. "Secure XML Views." In *Proceedings of the 16th IFIP WG11.3 Working Conference on Database and Application Security*. 2002.

[27] Trusted Computing Group. "TCG TPM Specification Version 1.2." TCG Specification.
URL `https://www.trustedcomputinggroup.org/specs/TPM`

[28] W3C (World Wide Web Consortium). "Document Object Model (DOM) Level 3 Core Specification." W3C Recommendation.
URL `http://www.w3.org/TR/DOM-Level-3-Core/`

[29] W3C (World Wide Web Consortium). "Extensible Markup Language (XML)." W3C Recommendation.
URL `http://www.w3c.org/XML/`