Fathom:
An Access Control-based Document Browsing Framework
Outline

Overview

Concepts
  Documents
  Viewing Policies
  Views and View Pointcuts

Fathom-AO: A Document Generator for Active Oberon
  Designing a Viewing Policy
Outline

Overview

Concepts
  Documents
  Viewing Policies
  Views and View Pointcuts

Fathom-AO: A Document Generator for Active Oberon
  Designing a Viewing Policy
What we are after

▶ To develop a framework in which the contents of a document can be navigated using mechanisms conceptually similar to the zoom and pan mechanisms used when viewing electronic representations of maps.

▶ To develop the topography of a document (and/or document type).

▶ To develop a pointcut-like language for specifying content within a document topography.

▶ To develop a security model enforcing access control over document content.
What we leave open

- We place modest/minimal restrictions on
  - the definition of the term *contents*.
  - the nature of a document *topography*.

- This enables Fathom to be *adapted* to a variety of *document types*. 
Definition

A document is modeled as a recursively structured composition of (need-to-know) categories and content.

The following operations are supported on the resulting model:

- **zoom** –
  - in the + direction brings content into the display range
  - in the – direction removes content from the display range

- **pan** – a mechanism used to position visible content in the display area.

- **need-to-know** – a mechanism for specifying (e.g., including or excluding) the categories upon which the zoom and pan functions operate.
Access Control

- We employ a \textit{security-based} subject/object model for specifying the \textit{visibility rights} that a \textit{view} (i.e., a subject) has over \textit{content} (i.e., an object). In order for a particular content \( x \) of a document to be visible in the display corresponding to a given view \( v \) the following is required:
  - \( v \) must have sufficient \textit{clearance} to access \( x \).
  - \( v \) must have a \textit{need-to-know} \( x \).
  - The \textit{zoom} must be sufficient to reveal \( x \).
  - The \textit{pan} is set properly (e.g., scroll-bar).
System Functions

▶ Off-Line:
  ▶ **Compiler** - A function that translates a document into an XML document model whose topology conforms to a particular viewing policy.
    ▶ A viewing policy is a specification of a document topography.

▶ Interactive:
  ▶ **Renderer** - A function that (in real-time) translates an XML document model into an HTML view according to a given selection criteria. Different selection criteria yield different HTML views.
  ▶ **Display** - A browser-like tool, tightly integrated with the renderer, for displaying HTML documents and related information (e.g., selection criteria).
Outline

Overview

Concepts

Documents
Viewing Policies
Views and View Pointcuts

Fathom-AO: A Document Generator for Active Oberon
Designing a Viewing Policy
Definition

- In Fathom, a document \( d \) is modeled as a tree consisting of \( \text{category} \) and \( \text{content} \) pieces.
- Henceforth, when we use the terms \( \text{document} \) and \( \text{category} \) we are referring to their tree representations.

Concretely, a document is compiled into an xml structure conforming to the following document type definition (DTD) fragment.

```
<!ELEMENT category (content | category)*>  
<!ELEMENT content (#PCDATA)>  
```
Example - Documents as Trees

Flat document

Tree structured document
### Definition

A viewing policy $\mathcal{P}$ is the formalism that we will use to specify:

- The *decomposition* of a document into categories and content.
- The *visibility attributes* assigned to content.

### Remark

*The objective of a viewing policy is to support a desired lattice of document views.*
A *viewing policy* is used to specify the visibility rights that *views* (interpreted as subjects) have over *contents* (interpreted as objects).

\[
\begin{array}{cccc}
  & c_1 & c_2 & \cdots & c_n \\
 v_1 & & & & \\
v_2 & & & & \\
\vdots & & & & \\
v_{2n} & & & & 
\end{array}
\]

It is not uncommon for \( n \) to be large. This presents a challenge with respect to (conceptually) populating the access control matrix shown above.
Clearance Rights

Document content is *assigned* a *clearance* level.

1 = public

2 = private

3 = secret
Document content is organized into a tree structure where content can be located at various depths.
Need-to-Know Categories

- Document content is assigned a need-to-know category.
- Document content is also positioned to fall within a specific category (which may be different from its assigned need-to-know category).

![Diagram of Need-to-Know Categories with depth levels 1, 2, and 3, marked with nodes and arrows.](image-url)
Need-to-Know Categories

▶ A document structure contains a finite number of need-to-know categories.

▶ We will use the symbol $C_d = \{\text{cat}_1, \ldots, \text{cat}_m\}$ to denote the set of category identifiers belonging to the document structure $d$.

▶ We will write $T_{\text{cat}_i}$ to denote the sub-tree in $d$ whose root node has the label $\text{cat}_i$. 
The elements of $C_d$ form a partial order with respect to the (node) ancestor relation. Specifically,

$$\text{cat}_i \preceq \text{cat}_j \overset{\text{def}}{=} (i = j) \lor \text{ancestor} (\text{cat}_i, \text{cat}_j).$$
In Fathom, need-to-know is Monotonic

- In Fathom, a *design decision* has been made requiring that *need-to-know rights* possessed by views be *monotonic*.

- This means that a need-to-know right *inherits* all of the need-to-know rights of its ancestors.
  - Rationale: *Zooming* in on part of a document should always “monotonically” increase the visible contents.

- Monotonicity of need-to-know rights becomes explicit in the definition of the *dominates* relation (which will be defined shortly).
Given document model $d$ having category set $C_d$, the visibility of content is determined by a triple of the form:

$$(\text{clearance}, \text{need-to-know}, \text{depth})$$

where

- $\text{clearance} \in \text{Integer}$
- $\text{need-to-know} \in C_d$
- $\text{depth} \in \text{Integer}$
Views

For a given document $d$, a view $v$ is a value taken from the following domain:

$$v \in \text{Integer} \times \mathcal{P}(C_d) \times \text{Integer}$$

The visibility of a view $v$ over a content element $c$ is determined by a dominates relation.

$$\text{isVisibleTo}(c, v) \overset{\text{def}}{=} v \text{ dominates } c$$
The Dominates Relation

- Adapted from the *Bell-LaPadula mandatory access control* mechanism.

- Let \( v \) and \( c \) respectively denote a document view and content element where:
  
  \[
  v = (clr_v, S, \text{depth}_v) \quad \text{where} \quad S \subseteq C_d
  \]
  
  \[
  c = (clr_c, \text{cat}_c, \text{depth}_c)
  \]

- The view \( v \) *dominates* \( c \) iff:
  
  \[
  clr_c \leq clr_v
  \]
  
  \[
  \exists \text{cat}_v \in S : \text{cat}_c \leq \text{cat}_v \land c \in T_{\text{cat}_v}
  \]
  
  \[
  \text{depth}_c \leq \text{depth}_v
  \]
The notation used to specify views can be generalized through the use of wildcard symbols such as \( \ast \). We refer to such a view as a view pointcut.

Whereas a view is associated with a single document, view pointcuts are applicable to sets of documents (e.g., they can be reused).

For example, a view pointcut can be constructed to display the interface or signature of an arbitrary software module or class.

Remark

*Depending on the level of expressivity desired, the view pointcut language can contain a variety of operators and matchable symbols.*
Outline

Overview

Concepts
  Documents
  Viewing Policies
  Views and View Pointcuts

Fathom-AO: A Document Generator for Active Oberon
  Designing a Viewing Policy
Active Oberon Documents

Definition

► *Fathom-AO* is an instance of Fathom in which

\[ D = \text{the set of applications written in Active Oberon.} \]

► In Fathom-AO, a document \( d = \{ m_1, \ldots, m_n \} \) corresponds to a set of Active Oberon modules.

► To create Fathom-AO the following must be accomplished:

  ► A *viewing policy* for AO applications must be designed.
  ► A *compiler* must be developed capable of transforming *AO applications* into *XML models* in a manner that conforms to the specified viewing policy.
Example: A View Sequence

MODULE M;

IMPORT
  M1, M2;

TYPE
  ptr1* = POINTER TO INTEGER;
  ptr2- = POINTER TO INTEGER;
  r1 = RECORD
        ... : INTEGER;
  y  : REAL;
  z- : INTEGER;
CONST
  a = 5;
  b* = 6;
  c- = 7;
BEGIN
  x := 1;
END M .
Use Case: Descriptions of Simple Views

Let \( d = \{ m_1, \ldots, m_n \} \).

**View** The names of all the modules in \( d \).

\[(\text{clearance}=0, \text{need-to-know}=\{\ast\}, \text{depth}=1)\]

**View** The top-level interfaces of all the modules in \( d \).

\[(\text{clearance}=1, \text{need-to-know}=\{\ast\}, \text{depth}=1)\]

**View** All exported elements for the modules in \( d \).

\[(\text{clearance}=1, \text{need-to-know}=\{m_1.\ast, \ldots, m_n.\ast\}, \text{depth}=\ast)\]

**View** The full-source of \( d \).

\[(\text{clearance}=\ast, \text{need-to-know}=\{m_1.\ast, \ldots, m_n.\ast\}, \text{depth}=\ast)\]
Use Case: Descriptions of Complex Views

View  The top-level interfaces corresponding to a given set of modules $m_j$ and $m_k$ in $d$.

$$(\text{clearance}=1, \text{need-to-know} = \{m_j, m_k\}, \text{depth}=1)$$

View  The private and public declarations for the module $m_i$.

$$(\text{clearance}=2, \text{need-to-know} = \{m_i\}, \text{depth}=1)$$

View  The top-level interfaces corresponding to the object $o_j$ within the module $m_i$.

$$(\text{clearance}=1, \text{need-to-know} = \{m_i.o_j\}, \text{depth}=2)$$
View  The top-level interface of $m_i$ plus the top-level interface corresponding to the object $o_j$ within $m_i$.

\[(\text{clearance}=1, \text{need-to-know} = \{m_i, m_i.o_j\}, \text{depth}=2)\]

View  The full contents of modules $m_i$ and $m_j$.

\[(\text{clearance} = *, \text{need-to-know} = \{m_i.*, m_j.*\}, \text{depth}=*)\]
Module Taxonomy

- **Categories**
  - Module bodies
  - Record bodies
  - Object bodies
  - Procedure bodies
  - Operator bodies

- Each category instance is assigned a (unique) need-to-know identifier.
Module Taxonomy

- **Content**
  - Module headers and footers
  - Import section
  - Record headers and footers
  - Object headers and footers
  - Procedure headers and footers
  - Type declarations excluding record and object declarations
  - Variable declarations
  - Constant declarations
  - Code

- Each *content type* is assigned a *clearance level* and a *need-to-know category*.

- Content resides within a *category* which resides at a fixed *depth* within the document.
Relative Need-to-Know: this

category

content

dashed arrow labeled "this"
Relative Need-to-Know: super
**Category** = Module

<table>
<thead>
<tr>
<th>Module Part</th>
<th>clearance</th>
<th>need-to-know</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>0</td>
<td>↓</td>
<td>1</td>
</tr>
<tr>
<td>import section</td>
<td></td>
<td>see import section</td>
<td></td>
</tr>
<tr>
<td>declarations</td>
<td></td>
<td>see declaration category</td>
<td></td>
</tr>
<tr>
<td>code</td>
<td></td>
<td>see code content</td>
<td></td>
</tr>
<tr>
<td>footer</td>
<td>public</td>
<td>this</td>
<td>1</td>
</tr>
</tbody>
</table>
Content = Import Section

<table>
<thead>
<tr>
<th>Import Content</th>
<th>clearance</th>
<th>need-to-know</th>
</tr>
</thead>
<tbody>
<tr>
<td>import section</td>
<td>public</td>
<td>this</td>
</tr>
</tbody>
</table>
## Content = Type Section

**Type Content**

<table>
<thead>
<tr>
<th></th>
<th>clearance</th>
<th>need-to-know</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td><strong>public</strong></td>
<td><strong>this</strong></td>
</tr>
<tr>
<td>record</td>
<td>see record category</td>
<td></td>
</tr>
<tr>
<td>object</td>
<td>see object category</td>
<td></td>
</tr>
<tr>
<td>array</td>
<td>see array content</td>
<td></td>
</tr>
<tr>
<td>pointer</td>
<td>see pointer content</td>
<td></td>
</tr>
<tr>
<td>ProcedureType</td>
<td>see ProcedureType content</td>
<td></td>
</tr>
<tr>
<td>QuanlifiedIdentifier</td>
<td>see QuanlifiedIdentifier content</td>
<td></td>
</tr>
</tbody>
</table>
Category = Record

### Public Record

<table>
<thead>
<tr>
<th></th>
<th>clearance</th>
<th>need-to-know</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>public</td>
<td>super</td>
</tr>
<tr>
<td>body</td>
<td>see variable declarations</td>
<td></td>
</tr>
<tr>
<td>footer</td>
<td>public</td>
<td>this</td>
</tr>
</tbody>
</table>

### Private Record

<table>
<thead>
<tr>
<th></th>
<th>clearance</th>
<th>need-to-know</th>
</tr>
</thead>
<tbody>
<tr>
<td>header</td>
<td>private</td>
<td>this</td>
</tr>
<tr>
<td>body</td>
<td>see variable declarations</td>
<td></td>
</tr>
<tr>
<td>footer</td>
<td>private</td>
<td>this</td>
</tr>
</tbody>
</table>
Content = Variable Declarations

<table>
<thead>
<tr>
<th>Public Variable Content</th>
<th>clearance</th>
<th>need-to-know</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration</td>
<td>public</td>
<td>this</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Private Variable Content</th>
<th>clearance</th>
<th>need-to-know</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration</td>
<td>private</td>
<td>this</td>
</tr>
</tbody>
</table>
Content = Code

<table>
<thead>
<tr>
<th>Code Content</th>
<th>clearance</th>
<th>need-to-know</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>secret</td>
<td>this</td>
</tr>
</tbody>
</table>
Demo