

## The *Fathom* System

Victor Winter  
Guest Professor - Native Systems Group  
ETH Zurich  
8092 Zurich, Switzerland  
Email: [vwinter@unomaha.edu](mailto:vwinter@unomaha.edu)

Ralf Lämmel  
Software Languages Team  
Universität Koblenz-Landau, Germany

Chandrasekhar Ramakrishnan  
Native Systems Group  
ETH Zurich  
8092 Zurich, Switzerland

## Abstract

This technical report describes a general-purpose approach to document generation and viewing. One begins by decomposing a document into *contents*. A *document view* is then defined as a subset of the contents of a document. The set of views for a given document can be related using the  $\subseteq$  operator. The resulting view-relation forms a lattice with the empty (document) view as the bottom element and the view containing the full document as the top element. Transitioning along the edges of this view-lattice is conceptually similar to the *zoom* mechanism used when viewing electronic representations of maps.

We have developed a document generation/viewing framework, called *Fathom*, based on the ideas described in the previous paragraph. Key challenges faced by Fathom include: (1) automatic decomposition of a document into a semantically meaningful content structure, (2) design of a suitable language for specifying views, and (3) automated support for navigating across an implied view-lattice (e.g., transitioning between views) and browsing the resulting sub-documents.

# Acknowledgements

I would like to thank all the members of the Native Systems group. Thanks to Ulrike (Ulla) Glavitsch Egger for setting up A2 on my system and for answering lots of AO questions. Thanks to Thomas Kaegi-Trachsel for helping me with all kinds of system stuff. Thanks to Fleix Friedrich and Florian Negele for providing me with a large AO code-base and for helping me better understand the theoretical underpinnings of AO. Thanks to Ling (Lisa) Liu for her idea on composing viewing policies. Thanks to Roman Mitin for showing me the document generation work that had been done for Zonnon. Thanks to Prof. Jürg Gutnkecht for his encouragement and for his ideas regarding the incorporation of module interconnectivity into document views. And thanks to Franziska Maeder and Martina Wirth for enabling me to spend my time on research problems rather than on administrative problems.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Philosophical Underpinnings . . . . .	5
1.2	Electronic Documents . . . . .	6
<b>2</b>	<b>Fathom</b>	<b>7</b>
2.1	Documents . . . . .	7
2.2	Document Content . . . . .	8
2.3	Document Categories . . . . .	9
2.4	Access Control . . . . .	9
2.4.1	The Dominates Relation . . . . .	11
2.5	View Specification . . . . .	12
2.6	Viewing Policies . . . . .	13
<b>3</b>	<b>Fathom-AO</b>	<b>15</b>
3.1	Designing a Viewing Policy . . . . .	15
3.1.1	Categories . . . . .	18
3.1.2	Content Units . . . . .	19
3.1.3	Comments . . . . .	19
3.1.4	Assigning Attributes to Content . . . . .	22
3.2	A Prototype of Fathom-AO . . . . .	25
<b>A</b>	<b>Future Work</b>	<b>28</b>
<b>B</b>	<b>Eight Equivalent Forms</b>	<b>30</b>
<b>C</b>	<b>Specification of a Viewing Policy</b>	<b>32</b>
<b>D</b>	<b>View Examples</b>	<b>35</b>
<b>E</b>	<b>A Compilation Example</b>	<b>38</b>

# List of Figures

2.1	Fathom Framework . . . . .	8
2.2	An XML document model . . . . .	10
2.3	The dom relation . . . . .	13
3.1	A BNF grammar fragment for Active Oberon declaration sequences	16
3.2	Semi-concrete examples of the topology of normalized categories	20
3.3	Semi-concrete examples of the topology of normalized TYPE, VAR, and CONST declarations . . . . .	21
3.4	The unformatted source code of the module M . . . . .	23
3.5	View sequences over the module M . . . . .	23
3.6	Denoting category labels in a relative fashion via <i>this_category/super_category</i>	25
3.7	A Tabular Specification of Records . . . . .	25
C.1	Viewing Policy Specification: Modules, Imports, and Type Decla- rations . . . . .	32
C.2	Viewing Policy Specification: Records, Objects and Variables . .	33
C.3	Viewing Policy Specification: Procedures and Operators . . . . .	34
D.1	Full . . . . .	35
D.2	Interface . . . . .	36
D.3	Signature . . . . .	36
D.4	Transitive . . . . .	37
E.1	The AO source of a small module. . . . .	38
E.2	FOO . . . . .	39

# List of Tables

2.1	Category Relationships for Figure 2.2 . . . . .	11
3.1	Summary of View Analysis . . . . .	18
3.2	View-pointcut examples for targeted and secondary views . . . . .	24

# Chapter 1

## Introduction

The goal of our research is to develop a framework, called Fathom, 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. In keeping with this metaphor, Fathom adapts a well-studied security model as the formalism for specifying the topography of a *document type* (i.e., the topography of a family of documents). A *document view* is then expressed using a pointcut-like language and provides the mechanism for specifying visibility rights over document contents. In addition, Fathom has been designed in such a fashion that a *(document) view* represents a document subset that is structurally consistent with respect to the document topology. This consistency provides for semantically smooth transitions between sequences of related views.

The purpose of navigation over a document topology is to facilitate informal/human comprehension of specific portions of the document. From a more general perspective, our aim is to develop a framework supporting interactive document *generation for* and informal *comprehension of* software systems. However, Fathom leaves relatively open the definition of *(document) contents* and the nature of a document topography. This enables the resulting framework to be adapted to a variety of document types.

### 1.1 Philosophical Underpinnings

In this section, we identify some key dimensions of documents in an abstract setting. These dimensions form the inspiration of the Fathom framework and its perspective towards document generation.

A *document*  $d_{1..n}$  can be modeled as a sequence of  $n$  information elements which we will refer to as *contents*.

$$d_{1..n} = \langle c_1, \dots, c_n \rangle$$

In this model, a content element  $c_i$  represents some meaningful unit of information. The position of a content element  $c_i$  within  $d_{1..n}$  may also convey information to the reader. In an ideal setting, the purpose of a particular content sequence  $d_{1..n}$  is to help the reader develop an understanding that lets them answer a set of question types:

$$\mathcal{Q} = \{q_1, \dots, q_k\}$$

In most cases, a sub-document provides a more concise representation of specific kinds of information. For this reason, in order to construct a document generator that provides an effective aid to human comprehension, it is essential to have an understanding of the types of questions users of the document generator system are interested in asking.

We assume that human memory is limited when compared to document size. Specifically, we assume that at any point in time a reader can only focus their attention (in a detailed manner) on a subset of the document. Thus, given a document  $d_{1..n}$  and a set of targeted/related questions  $\mathcal{Q}' \subseteq \mathcal{Q}$  the reader must form a subsequence of  $d_{1..n}$  for more detailed consideration. We use the term *document view* when referring to such subsequences. In an ideal setting, a (*document*) *view* should be minimal in the sense that it only contain document content germane to the targeted question set  $\mathcal{Q}'$ . Assuming the content sequence in a document  $d_{1..n}$  is ideal, it may also be useful to show (in some fashion) how the content in a *view* positionally relates to the content in  $d_{1..n}$ . This observation motivates the idea of supporting a zoom-like mechanism for document navigation. As one zooms in, more document content is revealed. As one zooms out, more content is hidden.

## 1.2 Electronic Documents

When  $d_{1..n}$  is stored in electronic form, there is a wide range of possibilities regarding the mechanisms that can be used to generate or otherwise support document views. When considered from content selection, search engines, and a variety of security models represent well-studied formalisms for selectively providing access to document content. From the perspective of content navigation, HTML links can be used to create associations between content along certain pre-determined dimensions of interest. Scroll bars can be used to traverse across large content sequences.

## Chapter 2

# Fathom

Fathom is a document generation and document viewing framework consisting of three components:

1. A *compiler* capable of translating a document, belonging to a given document family, into an XML model whose topology conforms to a particular viewing policy.
2. A *renderer* that interacts with a user and prepares for display document contents over which the current view has visibility rights.
3. A *display* integrated with the renderer which can be used to browse the contents visible to the given view.

A dataflow diagram of Fathom is shown in Figure 2.1. The benefits of this architecture is that only the compile stage is dependent upon domain-specific information. The render and display components can be reused across all document types. Thus, in order to adapt Fathom to a specific document type (e.g., a document generator/viewer for Java), only a compiler need be implemented.

### 2.1 Documents

In Fathom, a document  $d$  is modeled as a tree consisting of *category* and *content*. On an operational level, a document tree is an XML structure conforming to the following document type definition (DTD) fragment.

```
<!ELEMENT category (content | category)*>
<!ELEMENT content (#PCDATA)>
```

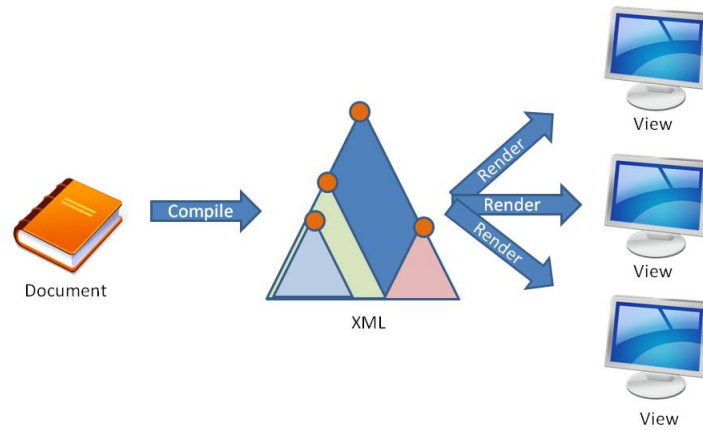


Figure 2.1: Fathom Framework

In general, the development of a compiler capable of translating a set of “flat documents” into their corresponding XML models is non-trivial. At the core of such a compiler is the identification of content units and the classification of content into categories. It should be noted that Fathom intentionally leaves open the definition of what constitutes a content unit. Fathom also leaves open the definition of categories. Within Fathom, these decisions are considered to be domain-specific and are to be specified by a viewing policy.

## 2.2 Document Content

Within a document, Fathom requires that each content element be assigned a *clearance-level* and a *need-to-know label*. Inherent to the XML document model is that content will fall at a given depth within a document tree (i.e., within a certain category). Thus, content *depth* represents an implied dimension along which content visibility may be controlled. Specifically, content visibility is a function of the triple (clearance, need-to-know, depth).

## 2.3 Document Categories

Fathom requires that a flat document be decomposed into a structure containing a finite number of *need-to-know categories*. We use the symbol  $\mathcal{C}_d = \{cat_1, \dots, cat_m\}$  to denote the set of category labels belonging to the document structure  $d$ . We write  $T_{cat_i}$  to denote the sub-tree in  $d$  whose root node has the label  $cat_i$ .

Fathom requires that a category label be fully qualified with respect to its position within the XML document structure. For example, if  $T_{cat_i}$  is a child of  $T_{cat_j}$  then the value of the label  $cat_i$  will contain the value of the label  $cat_j$  as a dot-separated prefix. Adopting a fully qualified naming convention is consistent with scoping rules found in mainstream programming languages. The use of fully qualified names also facilitates the automatic generation of unique and meaningful labels for large documents. And finally, as we will see in Section 2.5, such a naming convention supports a concise formulation of meaningful reusable views.

Figure 2.2 gives an example of a possible XML document tree model corresponding to the document  $d_{1..5} = \{c1, c2, c3, c4, c5\}$ . This XML model has  $\mathcal{C}_{d_{1..5}} = \{X1, X1.X2, X1.X3\}$  as its set of category labels. In the XML model, each content element has been translated into prettyprinted form, an HTML representation containing formatting information. Under these conditions, the task of Fathom’s render component is to extract, from the XML document model, the subset of prettyprinted contents over with a given view has visibility rights. The extracted elements are concatenated and packaged into an HTML document which is then presented to Fathom’s display component.

## 2.4 Access Control

The problem of controlling content visibility can be treated as a security problem. In particular, a view can be seen as a subject having *visibility rights* over specific content (seen as an object). This perspective opens the door to the consideration of a wide range of access control mechanisms. Fathom’s the render component implements a mandatory access control mechanism constraining access to document contents. This mandatory access control is based on a dominates relation between views and content. At the heart of this relation lies a partial order involving category labels. Equation 2.1 defines a *partial order* over the elements of  $\mathcal{C}_d$  based on the (node) ancestor relation.

$$cat_i \preceq cat_j \stackrel{def}{=} (i = j) \vee ancestor(cat_i, cat_j) \quad (2.1)$$

For example, the *preceq* relates the categories in Figure 2.2 as shown in Table 2.1.

```
<category id=X1>
  <content clearance=unclassified need_to_know=X1>
    prettyprint(c1)
  </content>
  <content clearance=classified need_to_know=X1>
    prettyprint(c2)
  </content>
  <category id=X1.X2>
    <content clearance=unclassified need_to_know=X1>
      prettyprint(c3)
    </content>
    <content clearance=classified need_to_know=X2>
      prettyprint(c4)
    </content>
  </category>
  <category id=X1.X3>
    <content clearance=classified need_to_know=X3>
      prettyprint(c5)
    </content>
  </category>
</category>
```

Figure 2.2: An XML document model

In Fathom, a design decision has been made requiring that need-to-know rights possessed by views be *monotonic*. Specifically, a need-to-know label subsumes all of the need-to-know labels of its ancestors. For example, in the XML model in Figure 2.2 the content element *prettyprint(c3)* has *X1* as its need-to-know label. Thus, *prettyprint(c3)* is a display candidate for any view having the need-to-know *X1*. However, since  $X1 \preceq X1.X2$ , *prettyprint(c3)* is also a display candidate for any view having the need-to-know *X1.X2*. Note that Fathom’s fully qualified category naming convention implies that a need-to-know label has viewing rights over any of its prefixes.

The rationale for monotonicity is that “zooming in” on a document should always result in a monotonic increase of the visible contents. Similarly, “zooming out” should always result in a monotonic decrease of the visible contents. Going back to our example of Figure 2.2, notice that *prettyprint(c3)* occurs in the category tree whose root is the label *X1.X2*. We believe that because of this structural relationship a view focusing on the contents of the category tree *X1.X2* with a

$X1$	$\preceq$	$X1$
$X1$	$\preceq$	$X1.X2$
$X1$	$\preceq$	$X1.X3$
$X1.X2$	$\preceq$	$X1.X2$
$X1.X3$	$\preceq$	$X1.X3$
$X1.X2$	$\not\preceq$	$X1.X3$
$X1.X3$	$\not\preceq$	$X1.X2$

Table 2.1: Category Relationships for Figure 2.2

need-to-know  $X1.X2$  should display at least as much contents as a view focusing on the the category tree  $X1$  with a need-to-know  $X1$ . The belief that the render-display component of Fathom should function this way is based on empirical evidence gathered when considering document generation and display for software systems.

#### 2.4.1 The Dominates Relation

The viewing domain  $\mathcal{V}$ , for a given document topology with category labels  $\mathcal{C}_d$ , is defined in Equation 2.2.

$$\mathcal{V} = Integer \times \mathcal{P}(\mathcal{C}_d) \times Integer \quad (2.2)$$

The *visibility* of a view  $v \in \mathcal{V}$  over a content element  $c$  is determined by a *dominates* relation. In particular,  $c$  is visible in  $v$  if and only if  $v$  *dominates*  $c$ . The *dominates* relation used by Fathom has been adapted from the *Bell-LaPadula mandatory access control* mechanism [1] and is formally defined as follows:

Let  $v$  and  $c$  respectively denote a document view and content element where:

- $v = (clr_v, S, depth_v)$  where  $S \subseteq \mathcal{C}_d$
- $c = (clr_c, cat_c, depth_c)$

The view  $v$  *dominates*  $c$  iff:

- $clr_c \leq clr_v$
- $\exists cat_v \in S : cat_c \preceq cat_v \wedge c \in T_{cat_v}$
- $depth_c \leq depth_v$

Note that in the above definition the condition  $c \in T_{cat_v}$  requires that visible content be located within a category tree over which the view has need-to-know rights. This restriction enables the specification of views whose contents are local to specific need-to-know category trees. That is, a view may only access the contents of category trees for which it has need-to-know rights. In particular, a view does not have universal access to all contents is its *preceq* relation. This localization capability is another reason motivating monotonic visibility along the need-to-know axis.

## 2.5 View Specification

Recall that a *view*  $\in \mathcal{V}$ , where  $\mathcal{V}$  is defined in Equation 2.2. In practice, requiring the construction of a view can be cumbersome. Especially when  $\mathcal{C}_d$  is large. For example, when considering document generation and viewing for software systems it is common to find (software) documents whose XML models contain thousands of categories. Thus, to remain effective in this environment, automated support must be provided for the construction of views.

To address this problem, the current version of Fathom supports a small pointcut-like language to facilitate the concise specification of views. In this language,  $\mathcal{V}$  is extended with a bottom category  $\perp$  and with a wildcard symbol  $*$ . To help the discussion, we will use the term *view-pointcut* when we wish to emphasize that view may contain symbols like  $\perp$  and  $*$ .

Presently, the use of the wildcard symbol is restricted. In particular, a category label within a view-pointcut may contain at most one wildcard and wild cards may not occur in the need-to-know labels associated with contents. Furthermore, if a category label contains a wildcard, the wildcard must occur in its final position. Given these restrictions the wildcard matching algorithm is straightforward. Suppose we want to compare the contents category label  $cat_c$  to a view-pointcut category label  $cat_v$  containing a wildcard. Let  $prefix(cat_v)$  denote the label obtained by dropping the wildcard from  $cat_v$ . If  $prefix(cat_v) \preceq cat_c$  then  $cat_v$  (the category with the wildcard) has need-to-know rights over  $cat_c$ . Note that in the case of wildcards, the  $\preceq$  comparison is reversed relative to the comparison used in the non-wildcard case. This reversal captures the ability of the wildcard to match with any portion of a category label.

Given a document  $d$  with categories  $\mathcal{C}_d$ . Let  $c = (clr_c, cat_c, depth_c)$  and let  $v = (clr_v, S, depth_v)$  where  $S \subseteq \mathcal{C}_d$ . The modified version of the dominates relation (which we refer to as *dom*) for view-pointcuts is given in Figure 2.3.

Within the aspect-oriented community, there is a large body of research that one can draw upon regarding the design of pointcut languages. Motivated by this body

$$v \text{ dom } c = \left\{ \begin{array}{l} (clr_c \leq clr_v \vee clr_v = *) \\ \wedge \\ ( \\ \quad cat_c = \perp \\ \quad \vee \\ \quad (\exists cat_v \in S : hasWild(cat_v) \rightarrow prefix(cat_v) \preceq cat_c \wedge c \in T_{cat_v}) \\ \quad \vee \\ \quad (\exists cat_v \in S : \neg hasWild(cat_v) \rightarrow cat_c \preceq cat_v \wedge c \in T_{cat_v}) \\ ) \\ \wedge \\ (depth_c \leq depth_v \vee clr_v = *) \end{array} \right.$$

Figure 2.3: The dom relation

of knowledge, our future research plans include an extension to the view-pointcut language presented here. Areas of particular interest include:

- An additional wildcard symbol capable of matching only a single element in a fully qualified category label. Such a symbol could occur multiple times within a category label-pointcut.
- An additional tag which could be added to labels. Such tags could convey type-like information about content. For example, in a software system we may want to specify the desire to view all procedure-related content or all procedure related content for procedures having a particular return type.
- The extension of the pointcut language to logical formulas (e.g., conjunctions, disjunctions and negations of expressions involving wildcard matches and comparisons).

## 2.6 Viewing Policies

A *viewing policy* is the formalism used to specify document topology. More specifically, a viewing policy specifies (1) the decomposition of a document into content and categories, and (2) the assignment of visibility attributes to content. In theory, a wide variety of notations could be considered as candidates for the specification of viewing policies. From such consideration a standard notation could then be selected. However, we believe that the suitability of a particular notation is depen-

dent upon the document type and compiler being developed. Thus, Fathom leaves open the notation used to specify a viewing policy and only identifies its goals.

## Chapter 3

# Fathom-AO

In this section, we demonstrate how Fathom can be adapted to *Active Oberon* [2], a concurrent extension of the Oberon language [6]. We use the term *Fathom-AO* when referring to the resulting framework. Before we begin, we would like to mention that the merits of a particular viewing policy are orthogonal to the merits of Fathom. That is to say, the critique of a viewing policy is insufficient to imply a critique of the Fathom framework as a whole. In order to critique Fathom it would be necessary to argue that the framework itself lacks expressive power or has some other inherent limitation.

### 3.1 Designing a Viewing Policy

For the Active Oberon (AO) domain, we define a document as a set of modules.

$$d_{1..n} = \{M_1, M_2, \dots, M_n\}$$

In practice, we are interested in sets of modules corresponding to AO applications and libraries. Our discussion on viewing policies for such documents centers primarily on controlling the visibility of declarations. Figure 3.1 shows a BNF grammar fragment highlighting the syntactic structure of AO's declaration sequences. The grammar fragment shows that recursively structured declaration sequences are possible within objects (i.e., `<ObjectType>`), procedures, and operators. Within a record it is possible for variable declarations to be recursively structured (e.g., a variable can be given an anonymous record type from within a record). Such recursive structures form the basis of our document topology. In particular, we would like to develop a viewing policy that enables use to zoom in on such structures.

<DeclarationSequence>	::=	<non_method_dec_list> <method_dec_list>
<non_method_dec_list>	::=	<non_method_dec> <non_method_dec_list>   $\epsilon$
<non_method_dec>	::=	<type_dec>   <var_dec>   <constant_dec>
<type_dec>	::=	TYPE <TypeDeclaration_list>
<TypeDeclaration_list>	::=	<TypeDeclaration> <TypeDeclaration_list>   $\epsilon$
<TypeDeclaration>	::=	<IdDef> = <Type> ;   ;
<var_dec>	::=	VAR <VariableDeclaration_list>
<VariableDeclaration_list>	::=	<VariableDeclaration> <VariableDeclaration_rest>
<VariableDeclaration_rest>	::=	; <VariableDeclaration> <VariableDeclaration_rest>   $\epsilon$
<VariableDeclaration>	::=	<VariableNameList> : <Type>   $\epsilon$
<VariableNameList>	::=	<VariableName> <VariableName_rest>
<VariableName_rest>	::=	, <VariableName> <VariableName_rest>   $\epsilon$
<constant_dec>	::=	CONST <ConstantDeclaration_list>
<ConstantDeclaration_list>	::=	<ConstantDeclaration> <ConstantDeclaration_list>
<Type>	::=	<ArrayType>   <PointerType>   <RecordType>   <ObjectType>   <ProcedureType>   <QualifiedIdentifier>
<ArrayType>	::=	ARRAY <Dimensions_opt> OF <Type>
<PointerType>	::=	POINTER TO <Type>
<RecordType>	::=	<record_header> <VariableDeclaration_opt> <record_footer>
<ObjectType>	::=	<object_header> <DeclarationSequence> <Body> <object_footer>
<VariableDeclaration_opt>	::=	<VariableDeclaration> <VariableDeclaration_rest>
<method_dec_list>	::=	<method_dec> <method_dec_list>   $\epsilon$
<method_dec>	::=	<ProcedureDeclaration>   <OperatorDeclaration>
<ProcedureDeclaration>	::=	<procedure_signature> <DeclarationSequence> <Body> <procedure_footer>
<OperatorDeclaration>	::=	<i>similar to procedure declaration</i>

Figure 3.1: A BNF grammar fragment for Active Oberon declaration sequences

In AO, type, variable, procedure and operator declarations can all be assigned attributes (similar to the public/private attributes in Java) controlling the scopes in which they may be referenced. A noteworthy semantic property of objects and records is that they can recursively export selected elements from within their structure. In our viewing policy, we are interested in creating document views that distinguish between such public and private declarations. In particular, we are interested in combining the ability to view this distinction with the ability to zoom in on recursive declaration structures. This motivates Definition 1 and Principle 1.

**Definition 1** *We define the [surface](#) of a declaration as follows: The header content represents the surface of a record or object, the signature represents the surface of a procedure or operator, and the entire declaration represents the surface of all other declarations. These concepts are further discussed in Section 3.1.2.*

**Principle 1** *If a view  $v$  does not have need-to-know rights over a declaration  $decl$ , then the visibility of  $v$  is limited to the surface of  $decl$ .*

As an initial step in designing a viewing policy it is essential to identify the kinds of views that are of primary importance. We refer to this set of views as the [targeted views](#). At a minimum a viewing policy must capture the set of targeted views. We initially target two views of interest: (1) a [full view](#), and (2) an [interface view](#).

The [full view](#) displays the full source of an AO document in pretty-printed form. It is a concrete embodiment of the recommended/standard formatting conventions for Active Oberon and may serve as an auto-formatting tool ensuring that the code produced by a given software development team consistently adheres to standard formatting conventions.

The [interface view](#) displays only (1) the imported and (2) the [surface](#) of exported declarations within an AO module. From a syntactic perspective, imported elements are declared using import statements. Exportable declarations include (1) type declarations, (2) variable declarations, (3) constant declarations, (4) procedure declarations, and (5) operator declarations.

In Section 3.2 we describe a prototype of Fathom-AO in which documents are transformed into a normal form where records have been flattened. However, no such transformation is considered for objects. What remains then is a flat record structure from which variables can be exported and a recursive object structure from which a variety of declarations can be exported. This motivates interest in a third view; a [transitive view](#) looking beyond the surface of declarations with visibility rights over all externally accessible elements within a structure, regardless of the nesting depth. (Note that this view is different from the interface view, which

targeted views	=	{ full, interface, signature, transitive, id }
secondary views	=	{ targeted views applied to sets of records, objects, procedures, and operators }

Table 3.1: Summary of View Analysis

only displays the surface of exported declarations and does not look further into the contents of records and active objects.)

It should be noted that an implementation of Fathom-AO providing these targeted views could also be extended to capture a number of additional views. Much of the infrastructure would already be in place. For this reason, it is worthwhile to identify a set of *secondary views* - views that are interesting but not essential.

For example, we may want to “apply” a targeted view to a selected subset of procedures, operators, records, and objects occurring anywhere within a module (e.g., the interface view could be applied to an object). This would provide a rich and flexible framework for browsing the contents of a document.

Along these lines, another view that may be interesting is to just show the surfaces of all declarations (public and private) within a module. This view essentially corresponds to the view provided by JavaDoc. For this reason, we add it to our set of targeted views. Our new targeted view is called a *signature view* and displays only the surface of a module’s declaration sequence. And finally, we also add to our targeted views a view showing just the header of a module (e.g., the keyword “MODULE” followed by the name of the module). We refer to this view as the *id view*. Table 3.1 summarizes the categorization of views resulting from our analysis.

### 3.1.1 Categories

The AO language has five constructs that our viewing policy will treat as constituting named scopes. These constructs are: (1) *modules*, (2) *procedures*, (3) *operators*, (4) named (active) *objects*, and (5) named (extensible) *records*. In Fathom-AO, these five constructs represent the need-to-know category types. Within a given document, instances of these category types are associated with identifiers. In AO, procedures, operators, and objects enable declaration sequences to be recursively nested. In Fathom-AO this translates into a recursive nesting of categories.

Fathom requires that category labels within a document be unique. In our implementation of Fathom-AO we construct category labels using fully-qualified paths with respect to category nesting. For example, a record named  $r$  within module  $M$  will have a category label  $M.r$ . This labeling policy assures that every named scope within an AO document will be given a unique category label.

### 3.1.2 Content Units

A *content unit* denotes an element of a document that our viewing policy treats as being atomic. As a result, visibility for a content unit with respect to a given view is a binary function; either the entire unit is visible or it is not. In our viewing policy, we define the following content units:

- the header information for a module, object, or record
- the signature information for a procedure or operator
- the footer information for a module, procedure, operator, object, or record
- the entire import section of a module
- each individual normalized pointer type declaration
- each individual normalized array declaration
- each individual normalized variable declaration
- each individual constant declaration
- code associated with a module, procedure, operator, or object

A semi-concrete example showing how categories are decomposed into content is given in Figure 3.2. A semi-concrete example showing how type, variable, and constant declarations are decomposed into content is shown in Figure 3.3. In these figures, nonterminal symbols and patterns from Figure 3.1 are used to abstract content units (e.g., `<Body>` in the case of the code unit, and `<VariableDeclaration>` in the case of a normalized variable declaration unit) or composite structures (e.g., `<DeclarationSequence>` and the category `<IdDef> = <RecordType>`).

### 3.1.3 Comments

In modern programming languages, the placement of comments within a program is largely unrestricted. As a result, such comments are not explicitly part of the grammar for a language, because the resulting grammar would then be ambiguous. Within a compiler, the lexical analysis phase typically handles comments. Specifically, comments are treated in a manner similar to white-space (i.e., they are read in and discarded during lexical analysis). However, for the purposes of document generation it is desirable to retain comments, at least in part.

The retention of comments beyond parsing presents some well-known problems. If all comments are to be retained and their placement within a program is

<b>Module Category</b>	
MODULE M;	← header unit
IMPORT M1, M2;	← import unit
<DeclarationSequence>	← composite
<Body>	← code unit
END M.	← footer unit
<b>Procedure Category</b>	
PROCEDURE p <FormalParameters_opt> ;	← signature unit
<DeclarationSequence>	← composite
<Body>	← code unit
END p;	← footer unit
<b>Operator Category</b>	
Operator “op” <FormalParameters_opt> ;	← signature unit
<DeclarationSequence>	← composite
<Body>	← code unit
END “op”;	← footer unit
<b>Object Category</b>	
o = OBJECT	← header unit
<DeclarationSequence>	← composite
<Body>	← code unit
END o;	← footer unit
<b>Record Category</b>	
r = RECORD	← header unit
<VariableDeclaration>	← variable declaration unit
END	← footer unit

Figure 3.2: Semi-concrete examples of the topology of normalized categories

unrestricted, then ambiguities arise regarding where exactly in a parse tree a comment should be placed. Another option, and one that is employed by document generators such as JavaDoc, is to only retain special comments and to restrict their placement within a program. This approach is also taken in Fathom-AO.

The Fathom-AO compiler uses a grammar for AO that has been extended to include *special comments*. A special comment may only occur immediately before

<b>Type Declaration Section</b>	
TYPE	← header unit
<IdDef> = <ArrayType>	← content unit
<IdDef> = <PointerType>	← content unit
<IdDef> = <RecordType>	← category
<IdDef> = <ObjectType>	← category
<IdDef> = <ProcedureType>	← content unit
<IdDef> = <QualifiedIdentifier>	← content unit
<b>Variable Declaration Section</b>	
VAR	← header unit
<VariableDeclaration>	← content unit
<b>Constant Declaration Section</b>	
CONST	← header unit
<ConstantDeclaration>	← content unit

Figure 3.3: Semi-concrete examples of the topology of normalized TYPE, VAR, and CONST declarations

the following AO constructs:

- A module
- An individual import element
- An individual constant declaration
- An individual variable declaration
- An individual type declaration
- An individual procedure declaration
- An individual operator declaration

In AO, an ordinary comment must begin with “(\*)” and end with “(\*)”. Fathom-AO requires that special comments begin with “(>)” and end with “(<\*)”. Furthermore, the contents of a special comment may not contain the symbol > (i.e., ASCII 60). Note that comments that are seen as special by Fathom-AO will be seen as ordinary comments by a standard AO compiler. This compatibility is important if Fathom-AO is to be compatible with other AO tools.

From the perspective of visibility, Fathom-AO associates special comments with the *surface* of corresponding AO elements. If the element is visible to a view, then so is the associated content. During rendering, special comments are simply copied verbatim into the resulting HTML document. This implies a number of limitations regarding the symbols that can occur with a special comment. Because of this, special comments represent an area of future work for Fathom-AO. In particular, special comments could be given a much richer syntactic and semantic structure. For example, JavaDoc provides a number of processable tags that may be used within a special comment.

### 3.1.4 Assigning Attributes to Content

At this point, we (1) have identified a set of targeted views as well as secondary views, and (2) have presented a semi-formal decomposition of AO documents into categories and content. What remains is the assignment of clearance and need-to-know attributes to content in a manner that is consistent with the targeted views and hopefully also the secondary views.

A useful first step in assigning clearance and need-to-know attributes to content is, for a given document, to construct a concrete view corresponding to each of the targeted views. In effect, this informally defines the semantics of views that we envision for the given document topology. Figure 3.4 gives an example of a document consisting of a single AO module. The module is named  $M$  and contains declarations for (1) a pointer  $ptr$ , (2) an object  $o$ , (3) a variable  $x$  and  $y$ , and (4) a procedure  $p1$  and  $p2$ . The object  $o$  is exported and also (transitively) exports the variable  $a$ . Also exported is the variable  $x$  and the procedure  $p1$ . Neither  $ptr$ ,  $y$  nor  $p2$  are exported.

Figure 3.5 is a concrete example showing how (1) a module is decomposed into content, and (2) the content visible to each of the targeted views.

Table 3.2 shows the view-pointcuts intended to capture the targeted and selected secondary views for the module  $M$ . These views assume that the integers denoting clearance levels have the following meaning: public=1, private=2, and secret=3. The difference between the interface and signature views implies that the surfaces of both exported and private declarations be located at depth 1 within the structure of  $M$  and that exported and private content be assigned clearances of 1 and 2 respectively.

It should be noted that additional views (which we have not listed) are automatically supported by the Fathom framework. For example, the view  $(2, \{M\}, 2)$  would display all public and private declarations in  $M$  – including the declaration of  $b$  in the object  $o$ . Also notice that the monotonic nature of views gives rise to interesting view sequences:  $id \rightarrow interface \rightarrow signature \rightarrow full$ . The Fathom-AO

```

MODULE M;
IMPORT M1;
TYPE ptr = POINTER TO INTEGER;
  o* = OBJECT VAR a*, b : INTEGER; END o;

VAR x*,y : INTEGER;

  PROCEDURE p1*() END p1;
  PROCEDURE p2() END p2;

BEGIN
  x := 1;
END M.

```

Figure 3.4: The unformatted source code of the module M

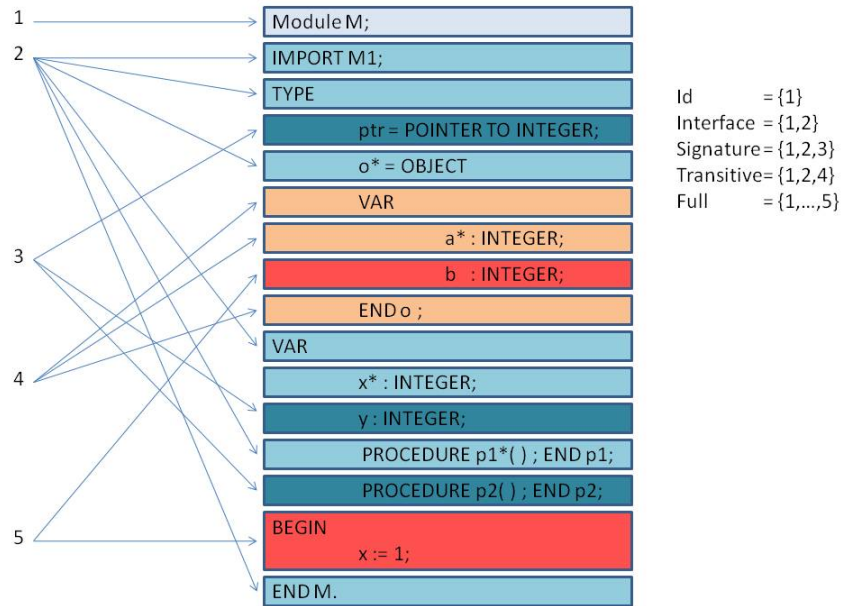


Figure 3.5: View sequences over the module M

view space forms a lattice with the empty view at the bottom and the full view at the top.

<b>Targeted Views</b>	
Id	= (1, { }, 1)
Interface	= (1, { M }, 1)
Signature	= (2, { M }, 1)
Transitive	= (1, { M.* }, *)
Full	= (3, { M.* }, *)
<b>Secondary Views</b>	
Id of object o	= (1, { M }, 2)
Interface of object o	= (1, { M.o }, 2)
Signature of object o	= (2, { M.o }, 1)
Transitive of object o	= (1, { M.o.* }, *)
Full of object o	= (3, { M.o.* }, *)

Table 3.2: View-pointcut examples for targeted and secondary views

Specifying the need-to-know label of a content unit in a relative manner is very useful given our document topology where category trees can be nested within one another. Figure 3.6 introduces two denotations that are of particular interest: (1) *this\_category* which for a content unit denotes the label of its immediately enclosing category tree, and (2) *super\_category* which denotes the parent category tree of the category tree having label *this\_category*. Similar denotations can be used to specify clearances: (1) *this\_clearance* denotes the clearance of *this\_category*, and (2) *super\_clearance* denotes the clearance of *super\_category*.

Using these denotations, a table-format can be developed capable of specifying AO viewing policies in a relatively formal manner. The idea is to create a set of tables accounting for all categories and content units and assign clearances and need-to-know labels to all table entries. The recursive nature of AO categories suggests that some table entries also have a recursive definition to them. Figure 3.7 gives an example of how visibility over records may be specified. A more comprehensive specification covering all categories and most contents is presented in Appendix C.

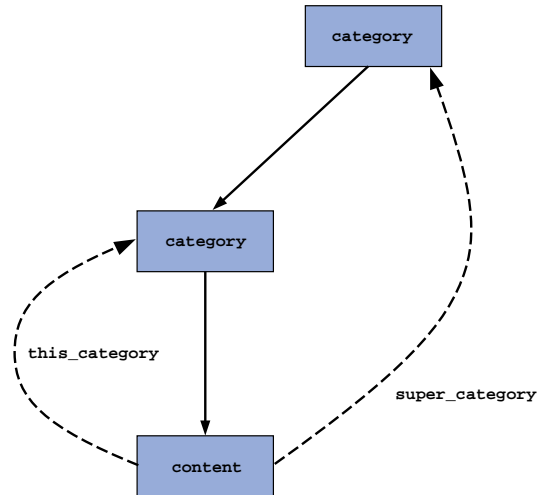


Figure 3.6: Denoting category labels in a relative fashion via *this\_category/super\_category*

#### Public/Private Record

	clearance	need-to-know
<record_header>	<i>public/private</i>	<i>super_category/this_category</i>
<VariableDeclaration_opt>	<i>see variable declarations</i>	
<record_footer>	<i>public</i>	<i>this_category</i>

#### Public/Private Variable Content

	clearance	need-to-know
<VariableDeclaration>	<i>public/private</i>	<i>this_category</i>

Figure 3.7: A Tabular Specification of Records

## 3.2 A Prototype of Fathom-AO

At present, we have transformation-based implementation of a prototype of Fathom-AO. The reason we call our implementation a prototype is because the rendering component has been decoupled from the display component and as a result is no longer interactive. In our implementation, rendering is strictly an off-line activity. Specifically, we have implemented both the compile and render components

of the Fathom-AO framework in HATS [3]. In our implementation, the compilation of an AO document consists of (1) a normalization phase followed by (2) a pretty-printing phase. In the normalization phase, AO modules are transformed in a variety of ways. A primary objective of normalization, which is essential to our viewing policy, is the removal of anonymous record types from variable declarations and type declarations. Conceptually, this results in a flattening of variable declarations and type declarations with respect to anonymous records. Somewhat related to this is a transformation that flattens variable lists within a variable declaration (e.g.,  $x, y : \text{INTEGER} \rightarrow x : \text{INTEGER}; y : \text{INTEGER}$ ). A secondary (non-essential) objective of normalization is the re-ordering of elements in a declaration list. In particular, a grouping is created where all type declarations occur first, followed by all variable declarations, followed by all constant declarations. The transformations in the normalization phase are implemented in the language TL [4][5] and are based on the axioms in Appendix B.

In the pretty-printing phase of the compiler the parse tree of a normalized document is formatted and converted into an XML document model. It is during this phase that contents are assigned clearance levels and need-to-know category labels. It is worth mentioning that within HATS, pretty-printing is functionally considered to be a restricted form of transformation. In particular, the HATS pretty-printer performs only a single top-down traversal of a parse tree and does so in a very efficient manner. In general, the primary goal of pretty-printing is to convert a parse tree into a formatted string. Conceptually, this is accomplished by inserting formatting information such as new-line symbols and indentation at appropriate places within the tree. However, the information that is inserted need not be limited to new-line symbols and indentation. In HATS, transformations written in TL have the property that they produce syntactically well-formed results with respect to a target grammar. This is enforced by TL. In contrast, the syntactic well-formedness requirement is relaxed during pretty-printing. In the pretty-printer, there are essentially no restrictions on the kind of string that may be produced. This makes the pretty-printer well-suited to for shifting between syntactic domains. For example, it is fairly straightforward to construct a pretty-printer capable of transforming a AO document into XML (or HTML). Within the pretty-printer, access to user-defined functions is also supported. Such functions provide access to a global state which may contain data structures such as stacks. For example, a category stack can be created tracking the nesting of named scopes. This stack can then be used to give a semantics to *this* and *super* as used in our viewing policy.

Recall that in Chapter 2 we mentioned that the rendering component of Fathom is domain-independent and can be reused across all instances of Fathom. In our prototype, we have implemented such a general-purpose rendering component in HATS. Our implementation takes an xml file as input, defines (in a hard-coded

manner) a selected number of views, and then produces HTML files corresponding to each of the selected views.

## Appendix A

# Future Work

- Ways could be explored to improve the XML document model.
  - For example, style sheets might be used to capture formatting information in a more elegant fashion than what is currently being done.
  - The information in an AO XML model could be increased. For example, HTML links could be added to connect content (e.g., the declaration of a type and its use in another declaration).
  - At present, only a single need-to-know label is assigned to content. In the original Bell-LaPadula dominates relation, the need-to-know component of content would consist of a set of labels. The implications of this shift could be explored and may lead to significant expressive enhancements in view expressivity.
- The viewing specification language could be significantly enriched.
  - For example, views could be expressed as boolean formulas.

$$\begin{aligned} &(\text{clearance} = 1 \wedge \text{need-to-know} = M) \\ &\vee \\ &(\text{clearance} = 2 \wedge \text{need-to-know} = M.p) \end{aligned}$$

- A variety of wildcard symbols could be added to the language (e.g., wildcards quantified over fully qualified paths versus wildcards quantified over individual labels, etc.). These wildcards could extend to HTML links. For example, Felix Fredrich had proposed the idea of a view that would show all uses of a particular (exported) type within an AO document. In this way, a view could essentially be used as a query mechanism for documents.

- Additional information, such as type information, could also be included in the attributes of content (e.g., as an extension/enrichment of the need-to-know information).
- A methodology for specifying viewing policies could be developed. A key capability, which may be possible in restricted settings, would be the automated composition of viewing policies. The idea of policy composition was initially suggested by Lisa Liu. Automated composition of viewing requirements would provide a highly interactive environment for crafting the most appropriate viewing policy.
- The information content of (preserved) comments could be dramatically improved. In particular, special comments could be given a much richer syntactic and semantic structure. For example, JavaDoc provides a number of processable tags that may be used within a special comment.
- Graphical elements could be used to convey structural relationships such as module inter-connectivity. This idea was proposed by Prof. Gutknecht.

## Appendix B

# Eight Equivalent Forms

In this section, we use the grammar fragment shown in Figure 3.1 as the basis for defining eight syntactic equivalences. Transformation of AO modules is based on these equivalences and produce AO documents whose structural properties are better suited for our viewing policy. In the listed axioms, subscripts are used to distinguish and associate nonterminal instances. The equivalences are based on the grammar fragment given in Figure 3.1 and are abstract in the sense that they gloss over some technical details. In particular, the syntactic composition of lists (e.g.,  $\langle \text{ConstantDeclaration\_list} \rangle_1 \langle \text{ConstantDeclaration\_list} \rangle_2$ ) is structurally imprecise. Ellipsis are also used as a mechanism to informally ignore certain structures. And finally, the subscripted nonterminal  $\langle \text{IdDef} \rangle_{\text{unique}}$  denotes a unique identifier.

A properly controlled application of the given AO-equivalences enables AO modules to be placed into a normal form well suited for Fathom-AO.

**Axiom 1** *Commutativity of non-method declarations.*

$$\begin{aligned} &\langle \text{non\_method\_dec} \rangle_1 \langle \text{non\_method\_dec} \rangle_2 \\ &\leftrightarrow \\ &\langle \text{non\_method\_dec} \rangle_2 \langle \text{non\_method\_dec} \rangle_1 \end{aligned}$$

**Axiom 2** *Absorption of the keyword CONST.*

$$\begin{aligned} &\text{CONST} \langle \text{ConstantDeclaration\_list} \rangle_1 \text{CONST} \langle \text{ConstantDeclaration\_list} \rangle_2 \\ &\leftrightarrow \\ &\text{CONST} \langle \text{ConstantDeclaration\_list} \rangle_1 \langle \text{ConstantDeclaration\_list} \rangle_2 \end{aligned}$$

**Axiom 3** *Absorption of the keyword VAR.*

$$\begin{aligned} & \text{VAR } \langle \text{VariableDeclaration\_list} \rangle_1 \text{ VAR } \langle \text{VariableDeclaration\_list} \rangle_2 \\ & \leftrightarrow \\ & \text{VAR } \langle \text{VariableDeclaration\_list} \rangle_1 \langle \text{VariableDeclaration\_list} \rangle_2 \end{aligned}$$

**Axiom 4** Absorption of the keyword *TYPE*.

$$\begin{aligned} & \text{TYPE } \langle \text{VariableDeclaration\_list} \rangle_1 \text{ TYPE } \langle \text{VariableDeclaration\_list} \rangle_2 \\ & \leftrightarrow \\ & \text{TYPE } \langle \text{VariableDeclaration\_list} \rangle_1 \langle \text{VariableDeclaration\_list} \rangle_2 \end{aligned}$$

**Axiom 5** Flattening a (complex) variable declaration.

$$\begin{aligned} & \text{VAR } \langle \text{VariableNameList} \rangle_1 : \langle \text{Type} \rangle_1 ; \\ & \leftrightarrow \\ & \text{TYPE } \langle \text{IdDef} \rangle_{\text{unique}} = \langle \text{Type} \rangle_1 ; \\ & \text{VAR } \langle \text{VariableNameList} \rangle_1 : \langle \text{IdDef} \rangle_{\text{unique}} \end{aligned}$$

**Axiom 6** Flattening an array type declaration.

$$\begin{aligned} & \text{TYPE } \langle \text{IdDef} \rangle_1 = \text{ARRAY } \langle \text{Dimensions\_opt} \rangle_1 \text{ OF } \langle \text{Type} \rangle_1 ; \\ & \leftrightarrow \\ & \text{TYPE } \langle \text{IdDef} \rangle_{\text{unique}} = \langle \text{Type} \rangle_1 ; \\ & \text{TYPE } \langle \text{IdDef} \rangle_1 = \text{ARRAY } \langle \text{Dimensions\_opt} \rangle_1 \text{ OF } \langle \text{IdDef} \rangle_{\text{unique}} \end{aligned}$$

**Axiom 7** Flattening a pointer type declaration.

$$\begin{aligned} & \text{TYPE } \langle \text{IdDef} \rangle_1 = \text{POINTER TO } \langle \text{Type} \rangle_1 ; \\ & \leftrightarrow \\ & \text{TYPE } \langle \text{IdDef} \rangle_{\text{unique}} = \langle \text{Type} \rangle_1 ; \\ & \text{TYPE } \langle \text{IdDef} \rangle_1 = \text{POINTER TO } \langle \text{IdDef} \rangle_{\text{unique}} \end{aligned}$$

**Axiom 8** Flattening a variable declaration within a record.

$$\begin{aligned} & \text{TYPE } \langle \text{IdDef} \rangle_1 = \text{RECORD } \dots \langle \text{VariableNameList} \rangle_1 : \langle \text{TYPE} \rangle_1 \dots \text{END}; \\ & \leftrightarrow \\ & \text{TYPE } \langle \text{IdDef} \rangle_{\text{unique}} = \langle \text{Type} \rangle_1 ; \\ & \text{TYPE } \langle \text{IdDef} \rangle_1 = \text{RECORD } \dots \langle \text{VariableNameList} \rangle_1 : \langle \text{IdDef} \rangle_{\text{unique}} \dots \text{END}; \end{aligned}$$

## Appendix C

# Specification of a Viewing Policy

### Module Category

	clearance	need-to-know
module_header	<i>public</i>	$\perp$
import section	<i>see import section</i>	
<DeclarationSequence>	<i>see declaration category</i>	
<Body>	<i>see code content</i>	
module_footer	<i>public</i>	<i>this_category</i>

### Import Content

	clearance	need-to-know
import section	<i>public</i>	<i>this_category</i>

### Type Declaration

	clearance	need-to-know
<IdDef> = <Type>	<i>public/private</i>	<i>super_category/this_category</i>

Figure C.1: Viewing Policy Specification: Modules, Imports, and Type Declarations

**Public/Private Record**

	clearance	need-to-know
<record_header>	<i>public/private</i>	<i>super_category/this_category</i>
<VariableDeclaration_opt>	<i>see variable declarations</i>	
<record_footer>	<i>public</i>	<i>this_category</i>

**Public/Private Object**

	clearance	need-to-know
<object_header>	<i>public/private</i>	<i>super_category/this_category</i>
<DeclarationSequence>	<i>see declarations</i>	
<Body>	<i>secret</i>	<i>this_category</i>
<object_footer>	<i>public</i>	<i>this_category</i>

**Public/Private Variable Content**

	clearance	need-to-know
<VariableDeclaration>	<i>public/private</i>	<i>this_category</i>

**Constant Section**

	clearance	need-to-know
CONST	<i>this_clearance</i>	<i>this_category</i>
<ConstantDeclaration_list>	<i>see &lt;ConstantDeclaration&gt;</i>	

**Public/Private Constant Content**

	clearance	need-to-know
<ConstantDeclaration>	<i>public/private</i>	<i>this_category</i>

Figure C.2: Viewing Policy Specification: Records, Objects and Variables

**Public/Private Procedure**

	clearance	need-to-know
<procedure_signature>	<i>public/private</i>	<i>super_category/this_category</i>
<DeclarationSequence>	<i>see declarations</i>	
<Body>	<i>secret</i>	<i>this_category</i>
<procedure_footer>	<i>public</i>	<i>this_category</i>

**Public/Private Operator**

	clearance	need-to-know
<operator_signature>	<i>public/private</i>	<i>super_category/this_category</i>
<DeclarationSequence>	<i>see declarations</i>	
<Body>	<i>secret</i>	<i>this_category</i>
<operator_footer>	<i>public</i>	<i>this_category</i>

**Code Content**

	clearance	need-to-know
<Body>	<i>secret</i>	<i>this_category</i>

Figure C.3: Viewing Policy Specification: Procedures and Operators

# Appendix D

## View Examples

```
MODULE M;
IMPORT
  M1;
TYPE
  ptr = POINTER TO INTEGER;
  o* = OBJECT
VAR
  a* : INTEGER;
  b : INTEGER;
END o ;
VAR
  x* : INTEGER;
  y : INTEGER;

  PROCEDURE p1*( ) ;
  END p1;

  PROCEDURE p2( ) ;
  END p2;

BEGIN
  x := 1 ;
END M.
```

Figure D.1: Full

```
MODULE M;
IMPORT
  M1;
TYPE
  ptr = POINTER TO INTEGER;
  o* = OBJECT
VAR
  x* : INTEGER;
  PROCEDURE p1*( ) ;
END M.
```

Figure D.2: Interface

```
MODULE M;
IMPORT
  M1;
TYPE
  ptr = POINTER TO INTEGER;
  o* = OBJECT
VAR
  x* : INTEGER;
  y : INTEGER;
  PROCEDURE p1*( ) ;
  PROCEDURE p2( ) ;
END M.
```

Figure D.3: Signature

```
MODULE M;
IMPORT
  M1;
TYPE
  o* = OBJECT
  VAR
    a* : INTEGER;
  END o;
VAR
  x* : INTEGER;
  PROCEDURE p1*( ) ;
  END p1;
END M.
```

Figure D.4: Transitive

## Appendix E

# A Compilation Example

```
MODULE M;  
  
VAR  
    x* : INTEGER;  
    y  : BOOLEAN;  
  
BEGIN  
    x := 1;  
END M .
```

Figure E.1: The AO source of a small module.



# Bibliography

- [1] L. LaPadula and D. Bell. Secure Computer Systems: Mathematical Foundations. Technical Report 2547, MITRE.
- [2] P. Reali. Active Oberon Language Report. Technical report, Institute für Computersysteme, ETH Zürich, 2004.
- [3] V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396, 2006.
- [4] V. Winter and M. Subramaniam. Dynamic Strategies, Transient Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.
- [5] V. L. Winter. Stack-based Strategic Control. In *Preproceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming*, June 2007.
- [6] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.