Integration of XML streams in information flow analysis for Java

Arnaud Fontaine

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

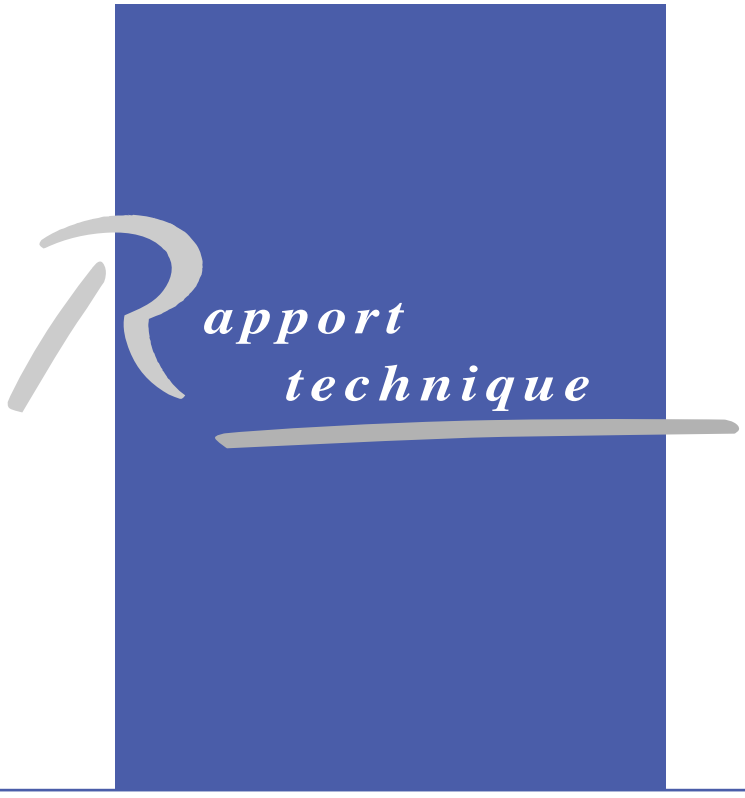
Integration of XML streams in information flow analysis for Java

Arnaud Fontaine

N° 0387

août 2010

Programs, Verification and Proofs



ISSN 0249-0803 ISRN INRIA/RT--0387--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
LILLE - NORD EUROPE

Integration of XML streams in information flow analysis for Java

Arnaud Fontaine

Theme : Programs, Verification and Proofs
Équipe-Projet POPS

Rapport technique n° 0387 — août 2010 — 37 pages

Abstract: In this report we present an extension of an existing flow-sensitive analysis for secure information flow for Java bytecode that deals with flows of data from and to XML streams governed by an access control mechanism. Our approach consists in computing, at different program points, an abstract XML content graph (AXCG) which tracks data read from and written to XML streams relying on data tracked in the existing information flow analysis. The extension we propose to manage XML content is generic enough to permit connection with any role-based access control mechanism for XML. On the contrary to many information flow techniques, our approach does not require security levels to be known during the analysis: security aspects of information flow and access control mechanisms for XML are checked *a posteriori* with security levels either inferred from access control policies for XML streams, or given by the information flow policy for the rest of the program.

Key-words: information flow, Java, XML, security, confidentiality

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Intégration des flux XML dans une analyse de flots d'information pour Java

Résumé : Nous présentons dans ce rapport une extension d'une analyse de flot d'information existante pour le bytecode Java qui permet d'intégrer les flots de données de et vers des flux XML soumis à un mécanisme de contrôle d'accès. Notre approche consiste à calculer, à différentes positions d'un programme, un graphe pour suivre les données lues et écrites dans les flux XML en se basant sur les flots de données suivis dans le modèle existant. L'extension que nous proposons est suffisamment générique pour permettre l'utilisation de n'importe quel mécanisme de contrôle d'accès pour XML de type RBAC. Contrairement à la plupart des techniques d'analyse flot d'information existantes, notre analyse ne nécessite pas l'assignement de niveaux de sécurité *a priori* : l'absence de fuite d'information est vérifiée *a posteriori* au regard des politiques de contrôle d'accès assignées aux flux XML et de la politique de sécurité relative aux flots d'information du reste du programme.

Mots-clés : flot d'information, Java, XML, sécurité, confidentialité

Contents

Introduction

Information flow issues for streaming have not yet been addressed by the community. In this report, we focus on XML streaming, a widely used technology in modern pervasive applications. Hence, it is now quite common within applications using external data to replace heavy databases systems by XML, which offer a light system to store information. XML files are used to store data of small or medium size such as application configuration files, user profiles, *etc*. A flow of information into a file or sent along a socket is considered as insecure for common static analysis; conversely, information read from files or sockets is usually considered as public. We aim to consider specific cases where such streams could be considered as safe with respect to information flow.

The solution we propose in this report is build on previous works on abstract interpretation through static analysis for Java bytecode. We keep track of the security levels considered in the program, and use an external access control model for XML for describing at which security level the information can be accessed. We combine the rich theory of information flow with the recent researches on access control for XML to obtain analysis where XML streams are properly taken into account. Read and write operations can be considered as dual with respect to information flow: when an information is recovered from a file, the security level attached to it must correspond to the level of all places in which it will be stored during the execution of the program; conversely, when a private/secure information is written in a file, it has to be done at a position that is considered/tagged by a high security level. One may expect from the program that it behaves safely according to this policy.

In Chapter 1 we first introduce some notations and the basis of access control mechanisms for XML required to understand encountered issues. We investigate in Chapter 2 the most common APIs used to read/write XML content in Java in order to choose the most appropriate to conduct an abstract interpretation in a static way, but also suitable for small constrained devices, which are the main targets of pervasive applications. Then, we describe in Chapter 3 how data read from and written to XML streams are tracked using *Abstract XML Content Graph (AXCG)*. Finally, we introduce in Chapter 4 *security oracles* that ensure the connection between the security levels of the existing information flow layer and access control flow policies attached to XML input/output streams.

Chapter 1

Preliminaries

The work presented in the following chapters of this report builds on previous work bearing on flow-sensitive analysis for Java bytecode [16, 21]. In this chapter we introduce in Section 1.1 some notations specific to the work detailed in this report. Then we give in Section 1.2 an overview of some relevant works.

1.1 Notations

The numerous definitions and notations introduced in [16, 21] are not reminded in this report. We complement this set of definitions and notations with some new additional definitions and notations used in the remaining parts of this report.

For a function f, we denote by $f\left[x \xmapsto{\phi(x)} y\right]$ the function f' such that $f'(x) = y$ if $\phi(x)$ holds and $f'(x) = f(x)$ otherwise.

We denote by $Chars$ the finite set of characters defined in UTF-8 charset[1]. We denote by $NonEmptyStrings$ the infinite set of string values of finite non-null length, *i.e.* the infinite set of characters tuples:

$$NonEmptyStrings = \{(c_0, \ldots, c_n) \mid n \in \mathbb{N}, 0 \leq i \leq n, c_i \in Chars\}$$

We denote by $Strings = NonEmptyStrings \cup \{\epsilon\}$ the infinite set of strings values of finite length where ϵ denotes the empty string (of length null). For simplicity, a string $s = (c_0, \ldots, c_n) \in Strings$ can also be written "$c_0 \ldots c_n$", and ϵ can be written "". We denote by $StringsPool$ the finite subset of $Strings$ extracted from the constant pools of all classes in the set $Classes$. Let string be a function to retrieve a string value from $StringsPool$ corresponding to an abstract constant value in a given AMG $G = (\overline{V}, \overline{E})$:

$$\mathsf{string} : \{\omega_i.\overline{cst}_j \mid \omega_i.\overline{cst}_j \in \overline{V}\} \longrightarrow StringsPool$$

We denote by $TagNames$ the finite subset of $Strings$ such that

$$TagNames = StringsPool \cup \{\mathsf{concat}(n, a) \mid n \in StringsPool, a \in StringsPool\}$$

[1] This charset is Java default charset for string internal representations.

where concat is the function defined as follows:

$$\begin{aligned}
\text{concat} : \mathit{StringsPool} \times \mathit{StringsPool} &\longrightarrow \mathit{Strings} \\
(\epsilon, \epsilon) &\longmapsto \epsilon \\
(\epsilon, (c_0^1, \ldots, c_n^1)) &\longmapsto (:, c_0^1, \ldots, c_n^1) \\
((c_0^1, \ldots, c_n^1), \epsilon) &\longmapsto (c_0^1, \ldots, c_n^1, :) \\
((c_0^1, \ldots, c_n^1), (c_0^2, \ldots, c_m^2)) &\longmapsto (c_0^1, \ldots, c_n^1, :, c_0^2, \ldots, c_m^2)
\end{aligned}$$

1.2 Related work

Secure information flow enforcement is a well studied area. A considerable amount of work on information flow control, based on static analysis and on theoretical foundations, has been achieved in the last decades. We choose not to detail secure information flow enforcement as it is discussed in articles and reports [16, 21] presenting the information flow analysis we rely on in the following chapters.

Access control mechanisms for XML is also a well studied area. In this section, we give only a short overview of access control mechanisms for XML. For a detailed survey, we encourage the reader to refer to [26, 12].

Historically, access control for XML is a topic strongly related to the database community [28, 14]. Most access control mechanisms for XML consequently inherit some of its paradigms from the definition of access control policies to their enforcement. Most access control mechanisms for XML follow the *Role-Based Access Control (RBAC)* model, described in Section 1.2.1, to define access control policies using *ad hoc* languages or a more standard language such as XACML [4] or XACL [23]. The enforcement of access control policies on XML documents can then be done in two ways described in Section 1.2.2: computing a *document view* or limiting access to the document via XPath (or another language) *queries* only.

1.2.1 Role-Based Access Control (RBAC)

Access control mechanisms that follow the *Role-Based Access Control (RBAC)* model [15, 34] rely on three sets of entities to define access control policies: *roles*, *objects* and *access modes*. A rule of an RBAC policy is simply a triplet $(role, object, access)$ that defines that $role$ can $access$ to $object$.

The *roles* set contains values describing who are the subjects of a rule. Roles can for instance map to a set of concrete users of a system, groups of users or (groups of) programs. The set of all available roles may be partially or totally ordered according to the semantics it is given. The mapping between roles and their corresponding concrete entities is a relevant problem that is not addressed here since this is an *authentication* problem.

The *objects* set is the critical part of RBAC, especially for XML documents and data. Indeed, the originality of a RBAC policy enforcement tool mostly resides in the expressiveness of the language used to describe objects. XPath [2, 3] is widely employed to describe XML elements as it is highly expressive and is a W3C standard.

The *access modes* set contains at least the “read” operation, *i.e.* read access to an object’s content. More refined read accesses also include “navigation” operation to control access to the structure of a protected XML document. Techniques devoted to XML-based databases also include “write” accesses such as deletion, insertion and updates of XML content, attributes and tags.

Frequent problems arising when trying to enforce RBAC policies is how to propagate rules and resolve conflicts between rules. Because of the tree structure of XML documents, it is mandatory to precisely define how a rule applying to a tag influences its attributes and child tags, and how to treat conflicts and contradictions between several applicable rules for the same set of objects.

1.2.2 Enforcement of access control policies for XML

Enforcement of an access control policy on an XML document can be achieved by computing a view of the document according to the requesting user [19, 34, 9, 8, 31, 5, 7, 13, 17, 29]. This method consists in pruning the document tree so as to contain only parts that the requesting user is allowed to access. The resulting document is referred to as *document view*. The main problem of this approach is that the document view may not be valid with the structure descriptor (DTD, XML schema, *etc*) of the original document, or an appropriate structure descriptor may need to be rebuild [25, 22, 27].

Enforcement of access control policies on XML content can also be achieved by secure querying instead of view computation [10, 18, 14, 28, 24, 30]. In this case, either non-violating (valid) queries are answered and result of applying the user query to the entire database.

We do not investigate further on both approaches as they are not appropriate in our context where raw documents must be directly accessed without, possibly heavy, computations such view computing or embedded query engine. In addition, none of these approaches deals with the enforcement of security policies on generated XML documents.

1.2.3 Information flow and XML data

At that time, we identified no work from the literature that tries to include readings and/or writings of XML documents with attached access control policies in an information flow analysis. Nevertheless, we identified some works [11, 33] focusing on semi-automatic object serialization into XML documents that could lead to elegant solutions. Access control policies attached to these documents could be automatically inferred from or injected to the information flow model. This approach is however limited since it could be impossible to deal with documents with an arbitrary structure, *i.e.* not mapped to/from objects hierarchy . To solve this issue, we could use on the fly XML transformations using the work of Castagna *et al.* [5] that deals with the enforcement of property equivalent to the non-interference property of information flow over XML transformations defined in CDuce [6] according to security labels inserted in the original XML document.

Chapter 2

Reading and writing XML in Java

Many ways can be used to read or write XML [1] in Java. However, not all these means are appropriate to study relevant information flow issues for small constrained Java-enabled devices. In this chapter we detail advantages and disadvantages of commonly used solutions in such context.

2.1 XML as flat text

In constrained environments, it is likely to read/write XML documents as flat text with procedures of standard input/output stream classes (namely `InputStream` and `OutputStream`). Since these classes are part of the Java runtime environment, no extra class (or library) needs to be deployed, which implies no additional memory consumption overhead. The main disadvantage of this mean is that it does not guarantee that XML content read or written is syntactically nor semantically correct. The programmer is in fact fully responsible to ensure these properties himself.

From the static information flow analysis point of view, dealing with characters sequences is not easy for several reasons. Difficulties encountered are actually exactly the same than those encountered with arrays of primitive values having different security levels: each sequence needs to be partitioned into parts with different security levels. This topic is still an open problem that is not adressed in this report [1]. Besides these difficulties, an XML content abstracted as a flat unidimensional structure is too rough to be able to later reason on tags scope and inclusion, and thus to permit any later connection with access control mechanisms for XML that mostly rely on the structure of XML documents.

2.2 XML API: DOM versus SAX

Besides simple text input/output streams, there exist many means to produce syntactically correct XML content, but also semantically valid with respect to a given grammar. Two different models are commonly used: DOM (*Document Object Model*), where XML documents are represented as trees, and SAX (*Simple Api for Xml*), where XML documents are

[1] Some refinements of static information flow analysis of arrays and hashed structures are investigated in the Work Package 4 of Sfincs project (`http://sfincs.gforge.inria.fr`) that also supports this work.

considered as sequences of opening/closing tags, *i.e.* a depth first traversal of XML tree structures.

The tree abstraction of DOM is the web-standard abstraction of XML documents (HTML, XHTML, *etc*) promoted by the W3C (*World Wide Web Consortium*). DOM is likely to be best suited for applications where the document must be accessed repeatedly and/or out of sequence order as it supports navigation in any direction. In fact, DOM paradigm implies a complete abstract representation of XML documents in memory before (resp. after) writing (resp. reading) them. This property makes DOM implementations not really appropriate in memory-constrained devices.

Implementations that follow the SAX model are generally preferred in constrained Java environments because they consume less memory than the DOM ones. XML elements are simply read and written sequentially without traceback facilities, which avoids the need to keep whole XML documents in memory.

The concept of the SAX approach is to produce a sequence of *events* (tag opening, tag closing, *etc*) corresponding to structural elements (enclosing tag, tag attributes,*etc*) encountered during the parsing of an XML document. Implementations of parsers that follow the SAX approach can be partitionned in two groups: event-driven implementations ("push-parsers"), where each event encountered fires a callback to a specific method, and lazy implementations ("pull-parsers"), where the programmer is in charge to enumerate the sequence of events. Between these two sets of implementations, lazy implementations are clearly more likely to be used for a static analysis simply because the structure of a read-/written XML document is not obfuscated by non predictable runtime callbacks. The structure of read/written XML documents clearly emerges in the code where lazy implementations are used, which simplifies static analysis and allow the inference of whole documents structures.

2.3 The XMLPull API

JavaSE and JavaME include a lazy SAX implementation (Sun XML API 1.0), but unfortunately signatures of its classes have not be generalized to make a clean API that could lead to easily substitutable alternative implementations. Moreover, this library has the same disadvantage as the StAX one: special events for comments and whitespaces occuring in XML documents that unnecessarily complicate lazy parsing. The most practical remaining libraries are implementations of the XMLPull API (version 1). Several implementations of the XMLPull API exist, among which the kXML (version 2), specially designed for JavaME environments, perfectly fits our requirements.

Basically, the XMLPull API defines two interfaces: a pull-parser (`XmlPullParser`) and a push-writer (`XmlSerializer`). The most useful methods of the parser are given Figure 2.2, including the five predefined constants that denote considered events encountered during parsing of XML documents. Forward iterations over an event sequence are controlled by the `next` method, while the `nextTag` method iterates over the super-sequence of opening and closing tags only. The `require` method is the most important one for the static analysis of parsed documents as its invocation only succeeds if the current event matches the criteria

given in parameters, and throws an exception if these criteria are not satisfied[2]. The two remaining methods `getText` and `getAttributeValue` are used to retrieve respectively a text content and an attribute value of the currently opened tag. Figure 2.4 illustrates how to use a `XmlPullParser` instance to parse the sample XML document of Figure 2.1. The most useful methods of the writer are given Figure 2.3. Each of these methods produces an XML event in the underlying output stream. These events naturally correspond to the events mentionned previously for the parser. The example displayed on Figure 2.5 illustrates how to use the `XmlSerializer` to generate the document of Figure 2.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore name="iBook store" size="2">
   <book nbauth="1">
      <title>Titre 1</title>
      <author>Auteur 1</author>
      <price>10</price>
   </book>
   <book nbauth="2">
      <title>Title 2</title>
      <author>Author 1</author>
      <author>Author 2</author>
      <price>20</price>
   </book>
</bookstore>
```

Figure 2.1: Example of XML content generated by the code displayed on Figure 2.5 and that can be parsed by the code displayed on Figure 2.4.

2.4 Contraints for static analysis purpose

Relying on the SAX approach, even with lazy implementations, is not sufficient to track data flows from/to XML documents in a pure static way. In fact, even if XML documents are sequentially read/written, this does not guarantee that the structure of the read/written XML document can be inferred, even partially. Some additional constraints must be applied on the use of methods from the XMLPull API in order to be able to detect XML markups and their scopes.

Lazy parsing of XML content using the XMLPull API strongly relies on the `require` method. We constrain its invocation with two pratical requirements. The first parameter (the expected type of the current event) must be one of the five constants (symbols, not concrete values) defined in the `XmlPullParser`. These constants are final static fields declared in this interface. However, our abstract interpretation model supports only non-static fields. For this reason, we reconsider the `XmlPullParser` interface to alter the definition of these fields from static to non-static, instance fields. The two remaining parameters of

[2]Only exceptions-free executions are considered from hypothesis of the original information flow.

```
/* Event constants */
static final int START_DOCUMENT,     /* first event */
                 START_TAG,          /* opening tag */
                 TEXT,               /* text content within opened tag (PCDATA) */
                 END_TAG,            /* closing tag */
                 END_DOCUMENT;       /* last event */

/* Sets the underlying input stream reader */
void setInput(Reader reader);

/* Enforces the event currently processed */
void require(int type,                          /* event type */
             String namespace,                  /* namespace of the event */
             String tag)                        /* local name of the tag */
             throws XmlPullParserException,
                    IOException;

/* Advances to the next event */
int next() throws XmlPullParserException, IOException;

/* Advances to the next tag opening or closing, skipping other events */
int nextTag() throws XmlPullParserException, IOException;

/* Reads the value of an attibute of the currently opened tag */
String getAttributeValue(String namespace,       /* namespace of the attribute */
                         String name);           /* local name of the attribute */

/* Reads the text content of the currently opened tag */
String getText();
```

Figure 2.2: Excerpt from the `XmlPullParser` interface declaration.

```
/* Sets the underlying output stream writer */
void setOuptut(Writer writer);

/* Opens a XML document for writing at URL location */
void startDocument(String enc,              /* charset encoding used */
                   boolean standalone)      /* is the document as an extrenal DTD ? */
                   throws IOException,
                          IllegalArgumentException,
                          IllegalStateException;

/* Writes an opening tag within the currently opened tag */
XmlSerializer startTag(String namespace,    /* namespace of the opening tag */
                       String name)         /* local name of the opening tag */
                       throws IOException,
                              IllegalArgumentException,
                              IllegalStateException;

/* Adds an attribute to the currently opened tag */
XmlSerializer attribute(String namespace,   /* namespace of the attribute */
                        String name,        /* local name of the attribute */
                        String value)       /* value to associate to this attribute */
                        throws IOException,
                               IllegalArgumentException,
                               IllegalStateException;

/* Writes some content to the currently opened tag */
XmlSerializer text(String text) throws IOException,
                                       IllegalArgumentException,
                                       IllegalStateException;

/* Writes the closing tag that corresponds to the currently opened tag */
XmlSerializer endTag(String namespace,      /* namespace of the closing tag */
                     String name)           /* local name of the closing tag */
                     throws IOException,
                            IllegalArgumentException,
                            IllegalStateException;

/* Closes a previously opened XMLOutputStream */
void endDocument() throws IOException,
                          IllegalArgumentException,
                          IllegalStateException;
```

Figure 2.3: Excerpt from the `XmlSerializer` interface declaration.

```
1  public void reader(String path) throws IOException,
2                                         XmlPullParserException {
3
4    XmlPullParser reader = new KXmlParser();
5    FileReader freader = new FileReader(path);
6    reader.setInput(freader);
7
8    reader.require(reader.START_DOCUMENT, null, null);
9
10   reader.nextTag(); reader.require(reader.START_TAG, null, "bookstore");
11   System.out.println("BOOKSTORE " + reader.getAttributeValue(null, "name"));
12   int nbbooks = Integer.parseInt(reader.getAttributeValue(null, "size"));
13
14   for(int i = 0; i < nbbooks; i++) {
15      reader.nextTag(); reader.require(reader.START_TAG, null, "book");
16
17      int nbaut = Integer.parseInt(reader.getAttributeValue(null, "nbaut"));
18
19      reader.nextTag(); reader.require(reader.START_TAG, null, "title");
20
21      reader.next(); reader.require(reader.TEXT, null, null);
22      System.out.println("BOOK + " + reader.getText());
23
24      reader.nextTag();
25      reader.require(reader.END_TAG, null, null); /* title */
26
27      for(int j = 0; j < nbauth; j++) {
28         reader.nextTag(); reader.require(reader.START_TAG, null, "author");
29
30         reader.next(); reader.require(reader.TEXT, null, null);
31         System.out.println("AUTHOR + " + reader.getText());
32
33         reader.nextTag();
34         reader.require(reader.END_TAG, null, null); /* author */
35      }
36
37      reader.nextTag(); reader.require(reader.START_TAG, null, "price");
38
39      reader.next(); reader.require(reader.TEXT, null, null);
40      System.out.println("PRICE + " + reader.getText());
41
42      reader.nextTag();
43      reader.require(reader.END_TAG, null, null); /* price */
44
45      reader.nextTag();
46      reader.require(reader.END_TAG, null, null); /* book */
47   }
48
49   reader.nextTag();
50   reader.require(reader.END_TAG, null, null); /* bookstore */
51
52   reader.next(); reader.require(reader.END_DOCUMENT, null, null);
53
54   reader.close(); freader.close();
55 }
```

Figure 2.4: Example of XML parsing in Java using kXML API.

```
1
2  public void writer(String path, Bookstore store) throws IOException,
3                                                          IllegalArgumentException,
4                                                          IllegalStateException {
5
6      Book[] books = store.books;
7
8      XmlSerializer writer = new KXmlSerializer();
9      FileWriter fwriter = new FileWriter(path);
10     writer.setOutput(fwriter);
11
12     writer.startDocument("UTF-8", null);
13
14     writer.startTag(null, "bookstore")
15     writer.attribute(null, "name", store.name);
16     writer.attribute(null, "size", books.length);
17
18     for(int i = 0; i < books.length; i++) {
19         Book book = books[i];
20
21         writer.startTag(null, "book").attribute(null, "nbaut", book.authors.size());
22
23         writer.startTag(null, "title").text(book.title).endTag(null, "title");
24
25         for(int j = 0; j < book.authors.length; j++) {
26             writer.startTag(null, "author");
27             writer.text(book.authors[j].name);
28             writer.endTag(null, "author");
29         }
30
31         writer.startTag(null, "price").text(book.price).endTag(null, "price");
32
33         writer.endTag(null, "book");
34     }
35
36     writer.endTag(null, "bookstore");     /* not mandatory */
37
38     writer.endDocument();
39
40     fwriter.close();
41 }
```

Figure 2.5: Example of XML generation in Java using kXML API.

the `require` method (the namespace and the name) must be plain string values (or null), *i.e.* that can be resolved into constants of the constant pool. The example displayed on Figure 2.4 fulfill all these constraints.

For XML content generation, methods signatures of the `XmlSerializer` class are explicit enough to determine the kind of structural element written. However, the same constraint on namespace and name parameters have to be applied. The example displayed on Figure 2.5 fulfill these constraints.

Chapter 3

XML content and information flow

In the previous chapter we have described some means to deal with XML content in Java. In this chapter we describe how to deal with XML data flows read/written with the XMLPull API in our static information flow model. We first describe in Section 3.1 a structure called *Abstract Xml Content Graph (AXCG)* to represent a set of XML documents in this context. Then, we detail in Sections 3.2 and 3.3 the modifications applied to our information flow model to include data flows from/to XML documents.

3.1 Abstract XML content graph

To catch the structure of parsed/written XML documents, we use a directed graph called *Abstract Xml Content Graph (AXCG)* build in parallel of the AMG during the analysis of a method's bytecode. Informally, vertices of an AXCG represent structural elements of XML: `Root` denotes the virtual root of a document, `Child` denotes an enclosing tag, `Text` denotes a text content within an enclosing tag, and `Attr` denotes an attribute of an enclosing tag. Edges of an AXCG describe arrangement of these structual elements. For instance, an edge from a `Child` vertex v to an `Attr` vertex a means that the attribute a is defined in the opening markup of v and is so an attribute of the tag v.

Since XML content, and *a fortiori* structural elements, can be parsed/written in loops whose bounds can be unknown during static analysis, the set of vertices of an AXCG is build by an approach equivalent to the allocation site model already used to build the set of vertices of an AMG. Each vertex is thus a pair of two values: the type of structural element (`Root`, `Child`, *etc*), and the position of the bytecode instruction where it is parsed/written.

The content of all documents read and written by the set of classes analysed is gathered in a single AXCG. To get the arrangement of XML elements within each XML document, each edge between two vertices is labelled with an abstract instance of parser/writer.

The link between the XML parts abstracted in an AXCG and abstract values from the original information flow analysis is ensured by the vertex labelling function η. Following the way flows of data are tracked in the existing information flow model, η associates to each vertex of the AXCG (given an abstract instance of parser/writer) two sets of vertices from the related AMG: the first one contains the abstract values of the tag name (only for `Child` and `Attr` vertices), and the second one abstract values its "content" depends on, explicitly or implicitly.

Definition 3.1.1 (Abstract XML Content Graph (AXCG))**.** *An* abstract XML content graph (AXCG) *is a finite directed graph $G_X = (\nu, \xi, \eta)$ build upon an AMG $\overline{G} = (\overline{V}, \overline{E})$ of a method $C_0.m_0$ for the call graph $CG = (V_{CG}, E_{CG})$ where $\nu = \nu_r \cup \nu_w$ is the set of vertices and $\xi = \xi_r \cup \xi_w$ is the set of edges with*

$$\nu_r \cup \nu_w \subseteq \{\texttt{Root}, \texttt{Child}, \texttt{Text}, \texttt{Attr}\} \times V_{CG}$$

$$\xi_r \subseteq \{((v_1, mpc_1), (v_2, mpc_2), \overline{o}) \mid (v_1, mpc_1) \in \nu_r \wedge (v_2, mpc_2) \in \nu_r \wedge \overline{o} \in \mathit{Readers}(\overline{V}) \\ \wedge ((v_1 = \texttt{Root} \wedge v_2 = \texttt{Child}) \vee (v_1 = \texttt{Child} \wedge v_2 \in \{\texttt{Child}, \texttt{Text}, \texttt{Attr}\}))\}$$

$$\xi_w \subseteq \{((v_1, mpc_1), (v_2, mpc_2), \overline{o}) \mid (v_1, mpc_1) \in \nu_w \wedge (v_2, mpc_2) \in \nu_w \wedge \overline{o} \in \mathit{Writers}(\overline{V}) \\ \wedge ((v_1 = \texttt{Root} \wedge v_2 = \texttt{Child}) \vee (v_1 = \texttt{Child} \wedge v_2 \in \{\texttt{Child}, \texttt{Text}, \texttt{Attr}\}))\}$$

$$\eta : (\mathit{Readers}(\overline{V}) \times \nu_r) \cup (\mathit{Writers}(\overline{V}) \times \nu_w) \longrightarrow \wp((\mathcal{V}(\overline{V}) \cup \{\overline{\texttt{null}}\}) \times \mathcal{V}(\overline{V})) \times \wp(\overline{V} \times \mathcal{F})$$

where $\mathit{Readers}(\overline{V}) = \{\overline{o} \mid \overline{o} \in \mathcal{O}(\overline{V}), \mathit{Type}(\overline{o}) \leq$ `XmlPullParser`$\}$ denotes the set of vertices of the AMG corresponding to instances implementing the `XmlPullParser` interface, and $\mathit{Writers}(\overline{V}) = \{\overline{o} \mid \overline{o} \in \mathcal{O}(\overline{V}), \mathit{Type}(\overline{o}) \leq$ `XmlSerializer`$\}$ denotes the set of vertices of the AMG corresponding to instances implementing the `XmlSerializer` interface.

In the same way as the set of AMG vertices, the set of AXCG vertices is built before abstract analysis of bytecode, *i.e.* application of transformation rules described in Section 3.3. `Root` vertices are "initial" vertices: no edge points to them. Reciprocally, `Attr` and `Text` vertices are "terminal" ones: there is no outgoing edge from these vertices. `Child` vertices can have both incoming and outgoing edges. For an inter-procedural call graph $V_{ICFG} = \{(\omega_i, v) \mid \omega_i \in V_{CG}, v \in V_{\mathsf{meth}(\omega_i)}\}$, the set of AXCG vertices is defined as follows:

$$\begin{aligned}\nu_w = &\{(\texttt{Root}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface startDocument}\} \\ &\cup \{(\texttt{Child}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface startTag}\} \\ &\cup \{(\texttt{Text}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface text}\} \\ &\cup \{(\texttt{Attr}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface attribute}\}\end{aligned}$$

$$\begin{aligned}\nu_r = &\{(\texttt{Root}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface require(START\_DOCUMENT,...)}\} \\ &\cup \{(\texttt{Child}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface require(START\_TAG,...)}\} \\ &\cup \{(\texttt{Text}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface getText}\} \\ &\cup \{(\texttt{Attr}, (\omega_i, j)) \mid (\omega_i, j) \in V_{ICFG}, P_{\mathsf{meth}(\omega_i)}[j] = \texttt{invokeinterface getAttributeValue}\}\end{aligned}$$

3.2 Abstract states

Abstract interpretation consists in computing the abstract states of the virtual machine before and after the execution of instructions of the analysed bytecode. The abstract state from the original model is extended to include AXCG's building.

Definition 3.2.1 (Abstract state)**.** *Let* $(\overline{Q}, R) = ((\overline{\rho}, \overline{s}, (\overline{V}, \overline{E})), (\delta_r, \delta_w, (\nu_r \cup \nu_w, \xi, \eta)))$ *be an abstract state where* $\overline{Q}$ *is the original abstract state part,* (ν, ξ, η) *is an AXCG, and* δ_r *(resp.* δ_w*) is a mapping that gives for each XML parser (resp. writer) the current position within the AXCG structure (last opened tags not yet closed).*

$$\delta_r : \mathit{Readers}(\overline{V}) \longrightarrow \wp(\{(v, mpc) \mid (v, mpc) \in \nu_r, v \in \{\texttt{Root}, \texttt{Child}\}\}$$
$$\delta_w : \mathit{Writers}(\overline{V}) \longrightarrow \wp(\{(v, mpc) \mid (v, mpc) \in \nu_w, v \in \{\texttt{Root}, \texttt{Child}\}\}$$

Two distinct mappings are used to mark the current position within a XML input/output stream to be able to deal with a single instance that implements both `XmlSerializer` and `XmlPullParser` interfaces.

To maintain the ascending chain property of the original model, the ordering relation and the join operation between abstract states have to be amended.

Definition 3.2.2 (Ordering relation $\sqsubseteq$ on property space $\mathcal{S}$)**.** *Let* $S_1 = (\overline{Q}_1, R_1, \Gamma_1) \in \mathcal{S}$ *and* $S_2 = (\overline{Q}_2, R_2, \Gamma_2) \in \mathcal{S}$ *be two abstract states with* $\overline{Q}_1 = (\overline{\rho}_1, \overline{s}_1, (\overline{V}_1, \overline{E}_1))$*,* $R_1 = (\delta_{r_1}, \delta_{w_1}, (\nu_1, \xi_1, \eta_1))$*,* $\overline{Q}_2 = (\overline{\rho}_2, \overline{s}_2, (\overline{V}_2, \overline{E}_2))$ *and* $R_2 = (\delta_{r_2}, \delta_{w_2}, (\nu_2, \xi_2, \eta_2))$*.* S_1 *is* smaller *than* S_2*, denoted* $S_1 \sqsubseteq S_2$*, if and only if* $\overline{Q}_1 \sqsubseteq \overline{Q}_2$*,* $R_1 \sqsubseteq R_2$ *and* $\Gamma_1 \subseteq \Gamma_2$*.* $R_1 \sqsubseteq R_2$ *holds iff* $\delta_{r_1} \sqsubseteq \delta_{r_2}$*,* $\delta_{w_1} \sqsubseteq \delta_{w_2}$*,* $\nu_1 \subseteq \nu_2$*,* $\xi_1 \subseteq \xi_2$ *and* $\eta_1 \sqsubseteq \eta_2$ *where*

$$\begin{aligned}
\delta_{r_1} \sqsubseteq \delta_{r_2} &\Leftrightarrow \forall \overline{o} \in \mathit{dom}(\delta_{r_1})\ \overline{o} \in \mathit{dom}(\delta_{r_2}) \wedge \delta_{r_1}(\overline{o}) \subseteq \delta_{r_2}(\overline{o}) \\
\delta_{w_1} \sqsubseteq \delta_{w_2} &\Leftrightarrow \forall \overline{o} \in \mathit{dom}(\delta_{w_1})\ \overline{o} \in \mathit{dom}(\delta_{w_2}) \wedge \delta_{w_1}(\overline{o}) \subseteq \delta_{w_2}(\overline{o}) \\
\eta_1 \sqsubseteq \eta_2 &\Leftrightarrow \forall (\overline{o}, (t, i)) \in \mathit{dom}(\eta_1)\ (\overline{o}, (t, i)) \in \mathit{dom}(\eta_2) \wedge \eta_1(\overline{o}, (t, i)) \sqsubseteq \eta_2(\overline{o}, (t, i))
\end{aligned}$$

and the relation $\sqsubseteq$ *in the formula* $\eta_1(\overline{o}, (t, i)) \sqsubseteq \eta_2(\overline{o}, (t, i))$ *is the component-wise application of the relation* $\subseteq$*.*

Definition 3.2.3 (Join operation $\sqcup$ on property space $\mathcal{S}$)**.** *Let* $S_1 = (\overline{Q}_1, R_1, \Gamma_1) \in \mathcal{S}$ *and* $S_2 = (\overline{Q}_2, R_2, \Gamma_2) \in \mathcal{S}$ *be two abstract states with* $\overline{Q}_1 = (\overline{\rho}_1, \overline{s}_1, (\overline{V}_1, \overline{E}_1))$*,* $R_1 = (\delta_{r_1}, \delta_{w_1}, (\nu_1, \xi_1, \eta_1))$*,* $\overline{Q}_2 = (\overline{\rho}_2, \overline{s}_2, (\overline{V}_2, \overline{E}_2))$ *and* $R_2 = (\delta_{r_2}, \delta_{w_2}, (\nu_2, \xi_2, \eta_2))$*. The* join *operation of* S_1 *and* S_2*, denoted* $S_1 \sqcup S_2$*, is defined as follows:*

$$\begin{aligned}
S_1 \sqcup S_2 &\Leftrightarrow (\overline{Q}_1 \sqcup \overline{Q}_2, R_1 \sqcup R_2, \Gamma_1 \cup \Gamma_2) \\
&\Leftrightarrow (\overline{Q}_1 \sqcup \overline{Q}_2, (\delta_{r_1} \sqcup \delta_{r_2}, \delta_{w_1} \sqcup \delta_{w_2}, \nu_1 \cup \nu_2, \xi_1 \cup \xi_2, \eta_1 \sqcup \eta_2), \Gamma_1 \cup \Gamma_2)
\end{aligned}$$

where

$$(\delta_{r_1} \sqcup \delta_{r_2})(\overline{o}) = \begin{cases} \delta_{r_1}(\overline{o}) \cup \delta_{r_2}(\overline{o}) & \text{if } \overline{o} \in \mathit{dom}(\delta_{r_1}) \cap \mathit{dom}(\delta_{r_2}) \\ \delta_{r_1}(\overline{o}) & \text{if } \overline{o} \in \mathit{dom}(\delta_{r_1}) \setminus \mathit{dom}(\delta_{r_2}) \\ \delta_{r_2}(\overline{o}) & \text{if } \overline{o} \in \mathit{dom}(\delta_{r_2}) \setminus \mathit{dom}(\delta_{r_1}) \end{cases}$$

$$(\delta_{w_1} \sqcup \delta_{w_2})(\overline{o}) = \begin{cases} \delta_{w_1}(\overline{o}) \cup \delta_{w_2}(\overline{o}) & \text{if } \overline{o} \in \mathit{dom}(\delta_{w_1}) \cap \mathit{dom}(\delta_{w_2}) \\ \delta_{w_1}(\overline{o}) & \text{if } \overline{o} \in \mathit{dom}(\delta_{w_1}) \setminus \mathit{dom}(\delta_{w_2}) \\ \delta_{w_2}(\overline{o}) & \text{if } \overline{o} \in \mathit{dom}(\delta_{w_2}) \setminus \mathit{dom}(\delta_{w_1}) \end{cases}$$

$$(\eta_1 \sqcup \eta_2)(\overline{o}, (t, i)) = \begin{cases} \eta_1(\overline{o}, (t, i)) \sqcup \eta_2(\overline{o}, (t, i)) & \text{if } (\overline{o}, (t, i)) \in \mathit{dom}(\eta_1) \cap \mathit{dom}(\eta_2) \\ \eta_1(\overline{o}, (t, i)) & \text{if } (\overline{o}, (t, i)) \in \mathit{dom}(\eta_1) \setminus \mathit{dom}(\eta_2) \\ \eta_2(\overline{o}, (t, i)) & \text{if } (\overline{o}, (t, i)) \in \mathit{dom}(\eta_2) \setminus \mathit{dom}(\eta_1) \end{cases}$$

and the operation $\sqcup$ *in the formula* $\eta_1(\overline{o},(t,i))\sqcup\eta_2(\overline{o},(t,i))$ *is the component-wise application of the operation* $\cup$*.*

3.3 Transformation rules

Evolutions from some constant initial state are dictated by the application of transformation rules, one for each bytecode instruction, until a fixed point is reached, *i.e.* abstract states at each bytecode position are unchanged by the application of transformation rules. The original set of rules is extended to build the AXCG, but rules that do not contribute or interfere with its build do not need to be modified: the new part added to an abstract state is simply copied as is. New transformation rules are added to specialize invocation of methods from the XMLPull API.

The two following sections introduce the transformations rules applying to XML writing (Section 3.3.1) and XML parsing (Section 3.3.2). In order to simply transformation rules definition we introduce the two following predicates and a function to deal with string values of tags and attributes names.

Definition 3.3.1 (Predicate pre)**.**

$$\mathsf{pre}((\nu,\xi,\eta),\delta,\overline{x},T) \triangleq \forall(\overline{o},\mathtt{d})\in\overline{x}\quad \emptyset\subset\delta(\overline{o})\subseteq\{(t,p)\mid t\in T,(t,p)\in\nu\}$$

Definition 3.3.2 (Predicate prematch)**.**

$$\begin{aligned}\mathsf{prematch}((\nu,\xi,\eta),\delta,\overline{x},T,\overline{na}) \triangleq\ & \mathsf{pre}((\nu,\xi,\eta),\delta,\overline{x},T)\\ \wedge\ & (\forall(\overline{o},\mathtt{d})\in\overline{x}\ \exists(c,\delta(\overline{o}),\overline{o})\in\xi)\\ \wedge\ & (\overline{na}=\{(\overline{\mathtt{null}},\overline{\mathtt{null}})\}\vee\\ & \quad\forall(\overline{o},\mathtt{d})\in\overline{x}\,\forall c\in\delta(\overline{o})\ (\overline{N},.)=\eta(\overline{o},c)\wedge\mathsf{qnames}(\overline{na})\subseteq\mathsf{qnames}(\overline{N}))\end{aligned}$$

Definition 3.3.3 (Function qnames)**.** *Let* qnames *be a function to build fully qualified names of XML tags and attributes (e.g.* `namespace:localname`*) from string values contained in constant pools.*

$$\begin{aligned}\mathsf{qnames} : \wp((\mathcal{V}(\overline{V})\cup\{\overline{\mathtt{null}}\})\times\mathcal{V}(\overline{V}))&\longrightarrow\wp(TagNames)\\ \overline{N}&\longmapsto\bigcup_{(\overline{n},\overline{a})\in\overline{N}}\mathsf{qnames}(\overline{n},\overline{a})\end{aligned}$$

$$\begin{aligned}\mathsf{qnames} : (\mathcal{V}(\overline{V})\cup\{\overline{\mathtt{null}}\})\times\mathcal{V}(\overline{V})&\longrightarrow TagNames\\ (\overline{n},\overline{a})&\longmapsto\begin{cases}\mathsf{string}(\overline{a}) & \text{if } \overline{n}=\overline{\mathtt{null}},\\ \mathsf{concat}(\mathsf{string}(\overline{n}),\mathsf{string}(\overline{a})) & \text{otherwise.}\end{cases}\end{aligned}$$

3.3.1 Transformation rules for XML writing

The Figure 3.1 contains the transformation rules corresponding to invocations of methods used to write XML content. The generic rules for method invocation is actually bypassed for these methods to introduce their new specific behaviour: the frame F of these rules

is left as is since their bytecode is not analysed. These methods have to be invoked on a valid instance of the class `XmlSerializer` with an underlying output stream set (method `setOutput`). If the reference to the instance of `XmlSerializer` is null, the underlying output stream has not been set or if it is not writable, an exception is raised and the output stream is untouched since it is assumed that no exception occurs during execution.

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{n}::\overline{e}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))\qquad \forall(\overline{o},\mathtt{d})\in\overline{x}\ \delta_w(\overline{o})=\emptyset\qquad m=\mathtt{startDocument(}\overline{e}\mathtt{,}\overline{n}\mathtt{)}}{(\overline{Q}',R')=((\overline{\rho},\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\{(\mathtt{Root},(\omega_i,j))\}\right],(\nu,\xi,\eta\left[(\overline{o},(\mathtt{Root},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Root},(\omega_i,j)))\sqcup(\emptyset,\overline{e}\cup\overline{n}\cup\Gamma)\right])))}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{a}::\overline{n}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))\qquad \forall(\overline{o},\mathtt{d})\in\overline{x}\ \delta_w(o)\neq\emptyset\qquad m=\mathtt{startTag(}\overline{n}\mathtt{,}\overline{a}\mathtt{)}}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\overline{x}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\{(\mathtt{Child},(\omega_i,j))\}\right],(\nu,\xi',\eta')))\\ \text{with }\xi'=\xi\cup\{(c,(\overline{v},(\omega_i,j)),\overline{o})\mid(\overline{v},(\omega_i,j))\in\nu,(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_w(\overline{o})\}\\ \text{and }\eta'=\eta\left[(\overline{o},(\mathtt{Child},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Child},(\omega_i,j)))\sqcup(\{(\overline{v}_1,\overline{v}_2)\mid(\overline{v}_1,\mathtt{d})\in\overline{n},(\overline{v}_2,\mathtt{d})\in\overline{a}\},\Gamma)\right]\end{array}}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{v}::\overline{a}::\overline{n}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))\qquad \mathsf{pre}(R,\delta_w,\overline{x},\{\mathtt{Child}\})\qquad m=\mathtt{attribute(}\overline{n}\mathtt{,}\overline{a}\mathtt{,}\overline{v}\mathtt{)}}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\overline{x}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi',\eta')))\\ \text{with }\xi'=\xi\cup\{(c,(\overline{v},(\omega_i,j)),\overline{o})\mid(\overline{v},(\omega_i,j))\in\nu,(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_w(\overline{o})\}\\ \text{and }\eta'=\eta\left[(\overline{o},(\mathtt{Attr},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Attr},(\omega_i,j)))\sqcup(\{(\overline{v}_1,\overline{v}_2)\mid(\overline{v}_1,\mathtt{d})\in\overline{n},(\overline{v}_2,\mathtt{d})\in\overline{a}\},\overline{v}\cup\Gamma)\right]\end{array}}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},v::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))\qquad \mathsf{pre}(R,\delta_w,\overline{x},\{\mathtt{Child}\})\qquad m=\mathtt{text(}\overline{v}\mathtt{)}}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\overline{x}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi',\eta\left[(\overline{o},(\mathtt{Text},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Text},(\omega_i,j)))\sqcup(\emptyset,\overline{v}\cup\Gamma)\right])))\\ \text{with }\xi'=\xi\cup\{(c,(\overline{v},(\omega_i,j)),\overline{o})\mid(\overline{v},(\omega_i,j))\in\nu,(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_w(\overline{o})\}\end{array}}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{a}::\overline{n}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))\qquad \mathsf{prematch}(R,\delta_w,\overline{x},\{\mathtt{Child}\},\overline{n}\times\overline{a})\qquad m=\mathtt{endTag(}\overline{n}\mathtt{,}\overline{a}\mathtt{)}}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\overline{x}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\{c\mid(c,\delta_w(\overline{o}),o)\in\xi\}\right],(\nu,\xi,\eta')))\\ \text{with }\eta'=\eta\left[(\overline{o},c)\overset{(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_w(\overline{o})}{\longmapsto}\eta(\overline{o},c)\sqcup(\{(\overline{v}_1,\overline{v}_2)\mid(\overline{v}_1,\mathtt{d})\in\overline{n},(\overline{v}_2,\mathtt{d})\in\overline{a}\},\Gamma)\right]\end{array}}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))\qquad \forall(\overline{o},\mathtt{d})\in\overline{x}\ \delta_w(\overline{o})\neq\emptyset\qquad m=\mathtt{endDocument()}}{(\overline{Q}',R')=((\overline{\rho},\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\emptyset\right],(\nu,\xi,\eta\left[(\overline{o},c)\overset{(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_w(\overline{o})}{\longmapsto}\eta(\overline{o},c)\sqcup(\emptyset,\Gamma)\right])))}$$

Figure 3.1: Abstract semantic rules XML output related methods where $(\overline{Q}',R')=instr^{(\omega_i,j)}_{\mathtt{invokeinterface\ XmlSerializer.}m}(\overline{Q},R,\Gamma)$.

The method `startDocument(`$\overline{e}$`,`$\overline{n}$`)` writes the header (`<?xml version=...`) of the XML document, with encoding $\overline{e}$ and namespace $\overline{n}$ applying to the whole content. This method should be called on instances that have not already been previously opened, *i.e.* $\delta_w(\overline{o})=\emptyset$ for each instance $\overline{o}$ on which it is potentially called. No edge needs to be created in the AXCG but the position in streams is set to the `Root` vertex of the AXCG for the current

bytecode instruction, and the label of this vertex is updated to include AMG vertices from $\overline{e}$ and $\overline{n}$, for each referenced output stream.

The method `startTag(`$\overline{n}$`,`$\overline{a}$`)` writes an opening tag named $\overline{a}$ in namespace $\overline{n}$ (if not null). As a result, an edge from the currently pointed vertex(vertices) to the `Child` vertex attached to the current bytecode instruction is added to the AXCG for each implied stream. The label of this `Child` vertex is also updated for each stream to include $\overline{n} \times \overline{a}$, couples of AMG vertices that denote its fully qualified name(s).

The method `attribute(`$\overline{n}$`,`$\overline{a}$`,`$\overline{v}$`)` writes a new attribute to the last opened tag(s) with name in $\overline{a}$, namespace in $\overline{n}$ (possibly null), and string value in $\overline{v}$. An edge from the currently pointed vertex(vertices) to the `Attr` vertex attached to the current bytecode instruction is added to the AXCG for each implied stream. The label of this `Attr` vertex is also updated for each stream to include $\overline{v}$, abstract values of the written content, as well as its fully qualified name(s) $\overline{n} \times \overline{a}$.

The method `text(`$\overline{v}$`)` writes the string value(s) in $\overline{v}$ to the last opened tag(s). An edge from the currently pointed vertex(vertices) to the `Text` vertex attached to the current bytecode instruction is added to the AXCG for each implied stream. The label of this `Text` vertex is also updated for each stream to include $\overline{v}$, abstract values of the written content.

The method `endTag(`$\overline{n}$`,` $\overline{a}$`)` writes a closing tag named $\overline{a}$ in namespace $\overline{n}$ (if not null). This method succeeds only if there is a last unclosed tag with the same name, that is why the name given to this closing tag is compared to all possible names of the last identified opened tag(s) for each related stream. In return, the position in each stream is updated to the corresponding parent tag(s).

The method `endDocument()` simply closes all unclosed start tags and flushes the underlying output stream. In order to reuse the `XmlSerializer` instances on which this method has been called, it is mandatory to call the method `setOutput` to reset the underlying output stream.

At the end of the analysis, when a fixed point is reached and the abstract state of the last method's instruction is correct, we can additionally check that all opened tags in XML output streams have been explicitly closed:

$$\forall \overline{o} \in \mathit{Writers}(\overline{V}) \quad \delta_w(\overline{o}) = \emptyset$$

The Figures 3.2 shows the AXCG resulting of the analysis of the method `writer` whose source code is given in Figure 2.5. The graph (a) shows the edges added during its analysis: there is only one instance of `XmlSerializer` in this method created at line 7[1] of this method abstracted by the AMG's vertex $\omega_0.\overline{o}_7$. On this graph clearly appears the tree structure of the generated XML content because the control flow of the method is very simple: opening tags and their corresponding closing tags are written in a non-ambiguous way, *i.e.* they are written in sequence in the same enclosing conditional branch. The graph (b) is the same graph where vertices have been replaced by their labels for the single `XmlSerializer` instance $\omega_0.\overline{o}_7$. On this graph, we can notice the second component of the label of the vertex $(\texttt{Text}, (\omega_0, 30))$. This set consists in three values that have to be interpreted as follows: $(\omega_0.\overline{p}_2.\texttt{books.price}, \texttt{d})$ denotes that the written content directly depends on the field

[1] For easier understanding, positions of bytecode instructions have been replaced by their related line numbers in the source code.

price of a Book instance stored in the array-field books of the second parameter of the method, $(\omega_0.\overline{p}_2.\texttt{books.length}, \texttt{i})$ denotes that the written content implicitly depends on the field length of a Book instance also stored in the array-field books of the second parameter of the method, and $(\omega_0.\overline{cst}_{18}, \texttt{i})$ denotes that the written content also implicitly depends on the constant appearing at line 18

3.3.2 Transformation rules for XML parsing

The Figure 3.3 contains the transformation rules corresponding to invocations of methods used to read XML content. As previously, the generic rule for method invocation is bypassed for these methods that have to be invoked on a valid instance of the class XmlPullParser with an underlying input stream set (method setInput).

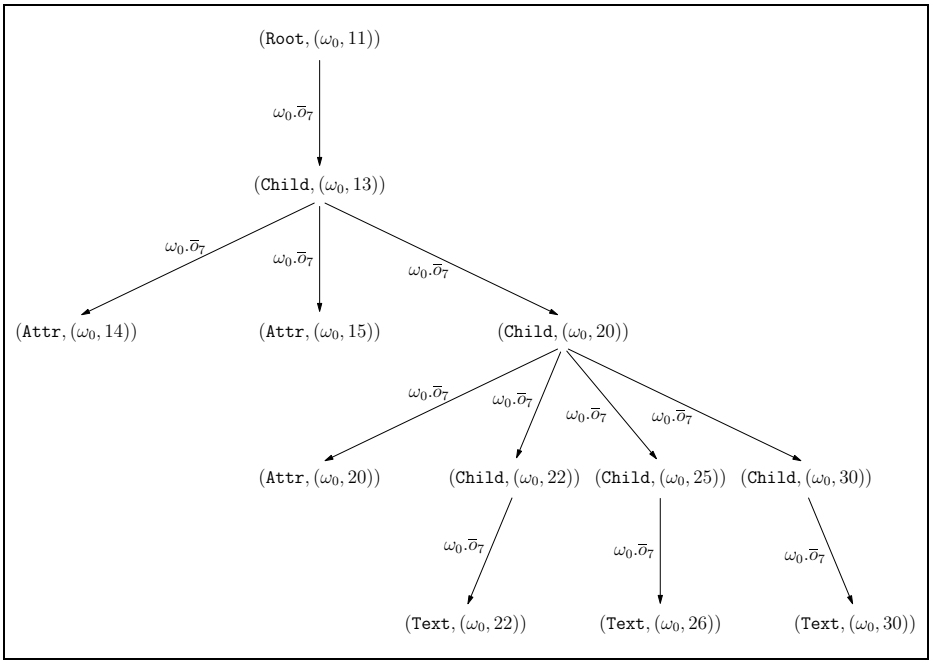
To infer the structure of parsed XML we strongly rely on invocations of the method require (Section 2.4). Several transformation rules are defined according to the value of the first parameter that denotes the expected kind of event. As it is assumed from hypothesis that method calls succeed without throwing exceptions, each encountered invocation of the method require permits to infer the existence of the structural element described by the first parameter value. No ambiguity is consequently permitted for the value of this parameter since it would result in intricate XML structures.

The method getAttributeValue($\overline{n}$,$\overline{a}$) permits to retrieve the string value of an attribute named $\overline{a}$ in namespace $\overline{n}$ (or null if outside a namespace) defined in the last currently opened tag, and puts the corresponding abstract value $\omega_0.\overline{o}_j$ on the top of the stack, or $\overline{\texttt{null}}$ if there is no such attribute. An edge from the currently pointed vertex(vertices) to the Attr vertex attached to the current bytecode instruction is added to the AXCG for each implied stream, and the label of this Attr vertex is also updated with the abstract string value returned to permit its *a posteriori* identification. The behaviour of the method getText() is exactly equivalent with Text vertices, except that no parameter is needed to identify the value to be retrieved.

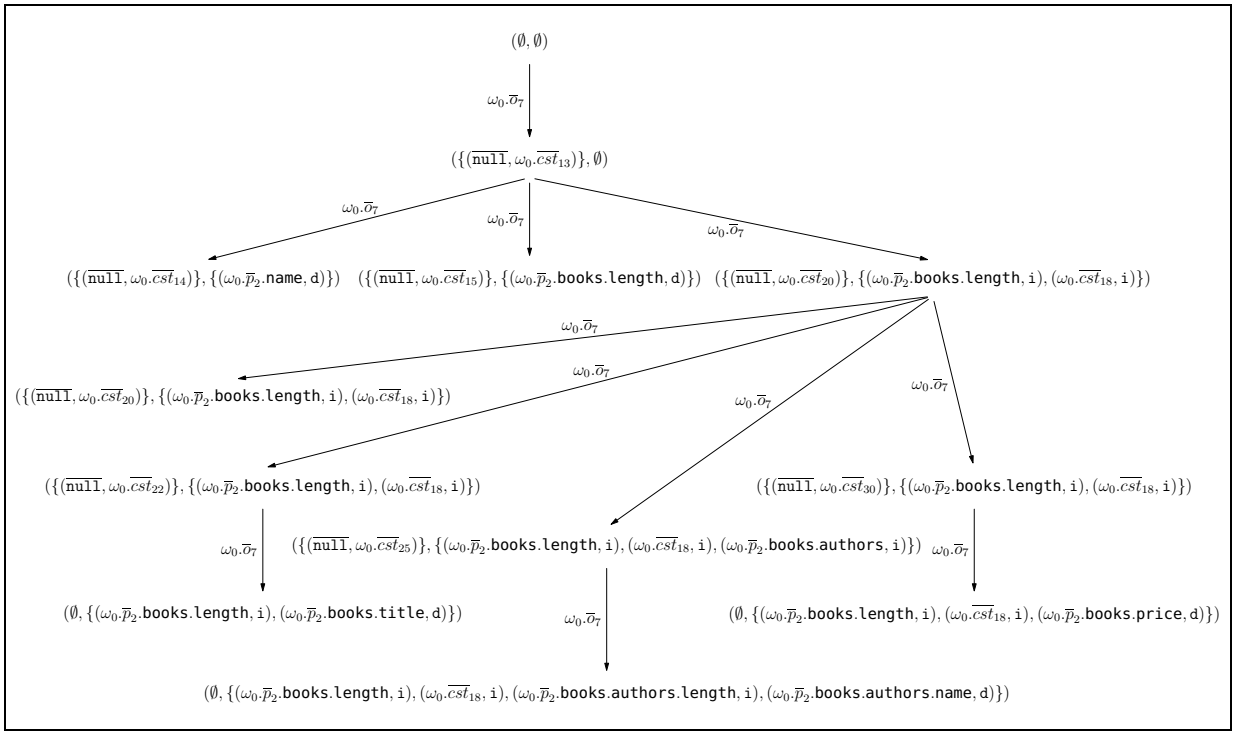
The methods next() and nextTag() have no impact on the AXCG. From the abstract interpretation point of view, they just return one of the constants defined in the XmlPullParser interface (Figure 2.2).

The Figures 3.4 shows the AXCG resulting of the analysis of the method writer whose source code is given in Figure 2.4. The graph (a) shows the edges added during its analysis: there is only one instance of XmlPullParser in this method created at line 4[2] of this method and abstracted by the value $\omega_0.\overline{o}_4$. On this graph clearly appears the tree structure of the generated XML content because the control flow of the method is very simple: opening tags and their corresponding closing tags are read in a non-ambiguous way, *i.e.* they are read in sequence in the same enclosing conditional branch. The graph (b) is the same graph where vertices have been replaced by their labels for the single XmlPullParser instance $\omega_0.\overline{o}_4$. On this graph, we can notice the second component of the label of the vertex $(\texttt{Text}, (\omega_0, 40))$. This set consists in three values that have to be interpreted as follows: $(\omega_0.\overline{o}_{40}, \texttt{d})$ denotes the string content read, $(\omega_0.\overline{o}_{12}, \texttt{i})$ denotes that the read content implicitly depends on an

[2] For easier understanding, positions of bytecode instructions have been replaced by their related line numbers in the source code.



(a) AXCG for the method $\mathsf{meth}(\omega_0) = \texttt{writer}$.



(b) AXCG where vertices have been replaced by their label for the stream $\omega_0.\overline{o}_7$ ($\forall x \in \nu_w \; x \mapsto \eta(\omega_0.\overline{o}_7, x)$).

Figure 3.2: Example of AXCG build for the code displayed on Figure 2.5. Bytecode instruction positions have been replaced by corresponding source code line numbers for easier understanding.

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{a}::\overline{n}::\overline{t}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \forall(\overline{o},\mathtt{d})\in\overline{x}\ \delta_r(\overline{o})=\emptyset \qquad \overline{t}_{\mathtt{d}}=\{\overline{o}.\mathtt{START\_DOCUMENT}\mid(\overline{o},\mathtt{d})\in\overline{x}\}}{(\overline{Q}',R')=((\overline{\rho},\overline{s},\overline{G},\overline{F}),(\delta_r\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\{(\mathtt{Root},(\omega_i,j))\}\right],\delta_w,(\nu,\xi,\eta\left[(\overline{o},(\mathtt{Root},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Root},(\omega_i,j)))\sqcup(\emptyset,\Gamma)\right])))}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{a}::\overline{n}::\overline{t}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \forall(\overline{o},\mathtt{d})\in\overline{x}\ \delta_r(\overline{o})\neq\emptyset \qquad \overline{t}_{\mathtt{d}}=\{\overline{o}.\mathtt{START\_TAG}\mid(\overline{o},\mathtt{d})\in\overline{x}\}}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\overline{s},\overline{G},\overline{F}),(\delta_r\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\{(\mathtt{Child},(\omega_i,j))\}\right],\delta_w,(\nu,\xi',\eta')))\\ \text{with }\xi'=\xi\cup\{(c,(\mathtt{Child},(\omega_i,j)),\overline{o})\mid(\mathtt{Child},(\omega_i,j))\in\nu,(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_r(\overline{o})\}\\ \text{and }\eta'=\eta\left[(\overline{o},(\mathtt{Child},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Child},(\omega_i,j)))\sqcup(\{(\overline{v}_1,\overline{v}_2)\mid(\overline{v}_1,\mathtt{d})\in\overline{n},(\overline{v}_2,\mathtt{d})\in\overline{a}\},\Gamma)\right]\end{array}}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{a}::\overline{n}::\overline{t}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \mathsf{prematch}(R,\delta_r,\overline{x},\{\mathtt{Child}\},\overline{n}\times\overline{a}) \qquad \overline{t}_{\mathtt{d}}=\{\overline{o}.\mathtt{END\_TAG}\mid(\overline{o},\mathtt{d})\in\overline{x}\}}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\overline{s},\overline{G},\overline{F}),(\delta_r\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\{c\mid(c,\overline{v},\overline{o})\in\xi,\overline{v}\in\delta_r(\overline{o})\}\right],\delta_w,(\nu,\xi,\eta')))\\ \text{with }\eta'=\eta\left[(\overline{o},c)\overset{(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_r(\overline{o})}{\longmapsto}\eta(\overline{o},c)\sqcup(\{(\overline{v}_1,\overline{v}_2)\mid(\overline{v}_1,\mathtt{d})\in\overline{n},(\overline{v}_2,\mathtt{d})\in\overline{a}\},\Gamma)\right]\end{array}}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{a}::\overline{n}::\overline{t}::\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \mathsf{pre}(R,\delta_r,x,\{\mathtt{Root}\}) \qquad \overline{t}_{\mathtt{d}}=\{\overline{o}.\mathtt{END\_DOCUMENT}\mid(\overline{o},\mathtt{d})\in\overline{x}\}}{(\overline{Q}',R')=((\overline{\rho},\overline{s},\overline{G},\overline{F}),(\delta_r\left[\overline{o}\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\emptyset\right],\delta_w,(\nu,\xi,\eta\left[(\overline{o},c)\overset{(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_r(\overline{o})}{\longmapsto}\eta(\overline{o},c)\sqcup(\emptyset,\Gamma)\right])))}$$

(a) Transformation rules for $m=\mathtt{require}(\overline{t},\overline{n},\overline{a})$ and $\overline{t}_{\mathtt{d}}=\{\overline{o}\mid(\overline{o},\mathtt{d})\in\overline{t}\}$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{a}::\overline{n}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \mathsf{pre}(R,\delta_r,x,\{\mathtt{Child}\}) \qquad m=\mathtt{getAttributeValue}(n,a)}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\{(\overline{\mathtt{null}},\mathtt{d}),(\omega_i.\overline{o}_j,\mathtt{d})\}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi',\eta')))\\ \text{with }\xi'=\xi\cup\{(c,(\mathtt{Attr},(\omega_i,j)),\overline{o})\mid(\mathtt{Attr},(\omega_i,j))\in\nu,(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_r(\overline{o})\}\\ \text{and }\eta'=\eta\left[(\overline{o},(\mathtt{Attr},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Attr},(\omega_i,j)))\sqcup(\{(\overline{v}_1,\overline{v}_2)\mid(\overline{v}_1,\mathtt{d})\in\overline{n},(\overline{v}_2,\mathtt{d})\in\overline{a}\},\{(\omega_i.\overline{o}_j,\mathtt{d})\}\cup\Gamma)\right]\end{array}}$$

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \mathsf{pre}(R,\delta_r,x,\{\mathtt{Child}\}) \qquad m=\mathtt{getText()}}{\begin{array}{c}(\overline{Q}',R')=((\overline{\rho},\{(\overline{\mathtt{null}},\mathtt{d}),(\omega_i.\overline{o}_j,\mathtt{d})\}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi',\eta')))\\ \text{with }\xi'=\xi\cup\{(c,(\mathtt{Text},(\omega_i,j)),\overline{o})\mid(\mathtt{Text},(\omega_i,j))\in\nu,(\overline{o},\mathtt{d})\in\overline{x},c\in\delta_r(\overline{o})\}\\ \text{and }\eta'=\eta\left[(\overline{o},(\mathtt{Text},(\omega_i,j)))\overset{(\overline{o},\mathtt{d})\in\overline{x}}{\longmapsto}\eta(\overline{o},(\mathtt{Text},(\omega_i,j)))\sqcup(\emptyset,\{(\omega_i.\overline{o}_j,\mathtt{d})\}\cup\Gamma)\right]\end{array}}$$

(b) Attribute and text content rules

$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \forall(\overline{o},\mathtt{d})\in\overline{x}\ \delta_r(\overline{o})\neq\emptyset \qquad m=\mathtt{next()}}{(\overline{Q}',R')=((\overline{\rho},\overline{V}_{\mathtt{doc}}\cup\overline{V}_{\mathtt{tag}}\cup\overline{V}_{\mathtt{text}}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))}$$

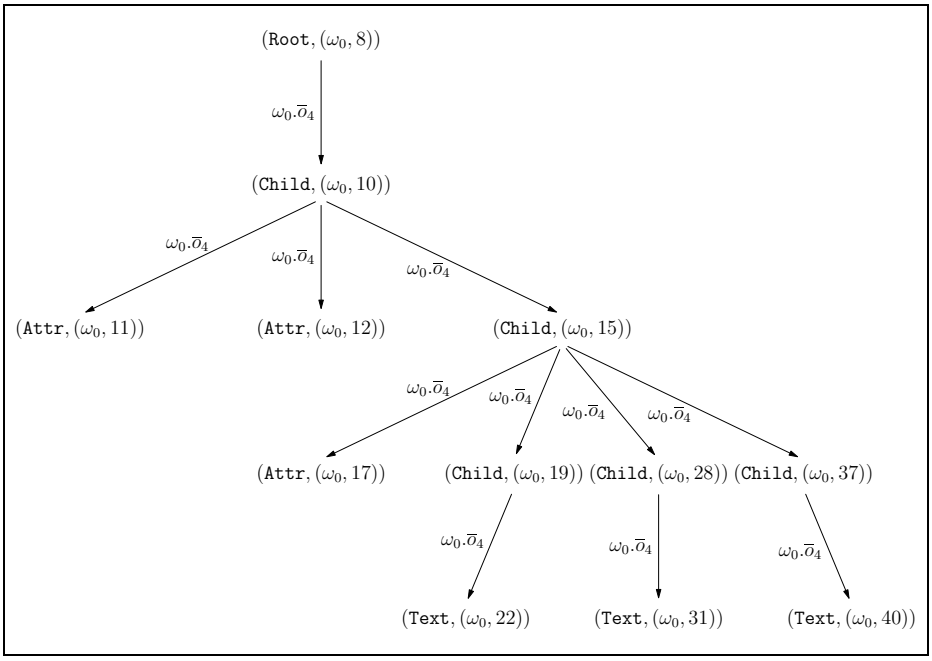
$$\frac{(\overline{Q},R)=((\overline{\rho},\overline{x}::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta))) \qquad \forall(\overline{o},\mathtt{d})\in\overline{x}\ \delta_r(\overline{o})\neq\emptyset \qquad m=\mathtt{nextTag()}}{(\overline{Q}',R')=((\overline{\rho},\overline{V}_{\mathtt{tag}}\cup\Gamma::\overline{s},\overline{G},\overline{F}),(\delta_r,\delta_w,(\nu,\xi,\eta)))}$$

$$\text{with}\quad \overline{V}_{\mathtt{doc}}=\{\overline{o}.\mathtt{START\_DOCUMENT},\overline{o}.\mathtt{END\_DOCUMENT}\mid(\overline{o},\mathtt{d})\in\overline{x}\} \qquad \overline{V}_{\mathtt{text}}=\{\overline{o}.\mathtt{TEXT}\mid(\overline{o},\mathtt{d})\in\overline{x}\}$$
$$\overline{V}_{\mathtt{tag}}=\{\overline{o}.\mathtt{START\_TAG},\overline{o}.\mathtt{END\_TAG}\mid(\overline{o},\mathtt{d})\in\overline{x}\}$$

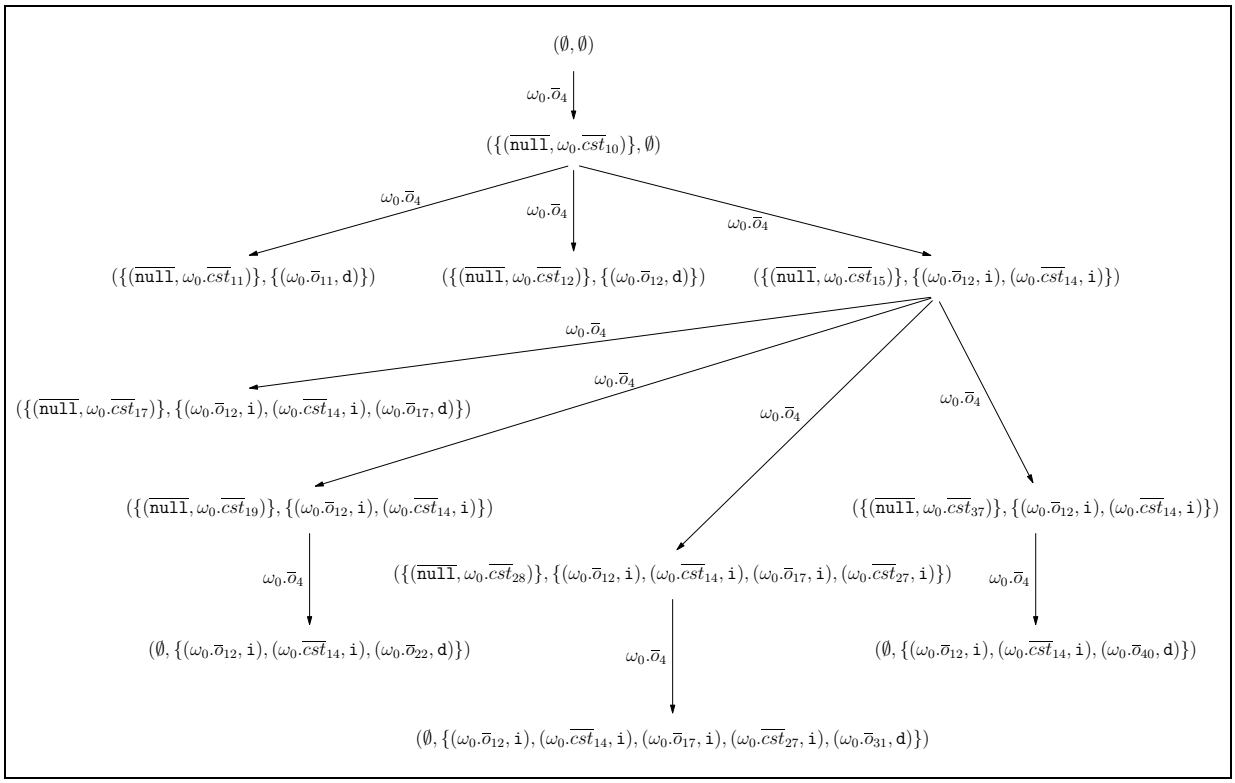
(c) Event forwarding functions

Figure 3.3: Transformation rules for invocation of methods from the `XmlPullParser` interface where $(\overline{Q}',R')=instr^{(\omega_i,j)}_{\mathtt{invokeinterface\ XmlPullParser}.m}(\overline{Q},R,\Gamma)$.

instance created at line 12 of this method, and $(\omega_0.\overline{cst}_{14}, \mathtt{i})$ denotes that the read content also implicitly depends on a constant appearing at line 14.



(a) AXCG for the method meth(ω_0) = reader.



(b) AXCG where vertices have been replaced by their label for the stream $\omega_0.\overline{o}_4$ ($\forall x \in \nu_w, x \mapsto \eta(\omega_0.\overline{o}_4, x)$).

Figure 3.4: Example of AXCG build for the code displayed on Figure 2.4. Bytecode instruction positions have been replaced by corresponding source code line numbers for easier understanding.

Chapter 4

Secure information flow and AXCG

The AXCG resulting of the abstract analysis of a method contains all the links between XML content read/written through the XML API and values abstracted in the related AMG. In this chapter we extend the original application of the information flow analysis to secure information flow to connect it to access control mechanisms for XML. Actually, we focus on security policies that apply only to XML documents instances and not to their structures. We describe in Section 4.1 an abstract way to connect our extended information flow model to access control mechanisms for XML based on XPath descriptors, and illustrate how this can be instantiated for a concrete access control mechanism in Section 4.2

4.1 Interface between information flow policy and access control for XML

Many definitions of (secure) information flow exist in the literature from traditional non-interference from the earliest years to notions related to software engineering like [32]. The basic idea of all the approaches is to define a lattice of security levels and to control the flows of data between the different security levels. Previous works on non-interference require the lattice of security levels and the model for security policy to be known from the beginning. This requirement is avoided with AMG since it contains points-to and dependency between data, and can thus be labeled by security levels and non-interference checked *a posteriori*. The main advantage of this approach is to support changes of security levels without reanalyzing programs.

Since AXCG reuses AMG's vertices to keep track of data flows from/to XML streams, it is natural to extend to AXCG the scheme of labelling used for AMG with the same security lattice $(\mathcal{L}, \sqcup, \sqsubseteq)$. For the purpose of connecting secure information flow and access control mechanisms for XML, we introduce security oracles. We describe the type of security oracle functions and introduce several facilities to instantiate these oracles, such as a simple way to practically set the access control policy of an XML parser/serializer and an algorithm to identify AXCG vertices matching a XPath descriptor.

4.1.1 Security oracles

Security oracle for data written to XML streams

Attribute and content values written to XML streams identified in the AXCG have security levels coming directly from the information flow model and its security policy. The security level of an AXCG vertex, given by the function λ_{ν_w}, is inferred using the least upper bound of security levels of abstract values representing its content.

Definition 4.1.1 (Security function λ_{ν_w})**.** *Let λ_{ν_w} be a function that assigns a security level to the vertices of the AXCG related to XML output streams using security levels assigned to AMG's vertices by the function λ_v [16, 21].*

$$\lambda_{\nu_w} : \mathit{Writers}(\overline{V}) \times \nu_w \longrightarrow \mathcal{L}$$
$$(\overline{o}, v) \longmapsto \bigsqcup_{(\overline{N},\overline{C})\in\eta(\overline{o},v),\overline{x}\in\{\overline{n},\overline{a}|(\overline{n},\overline{a})\in\overline{N}\}\cup\{\overline{c}|(\overline{c},.)\in\overline{C}\}} \lambda_v(\overline{x})$$

We have to ensure that the security level inferred for each written data is coherent with the expected security level coming from its related access control policy. The expected security level of an AXCG vertex is given by the security oracle function

$$\Omega_w : \mathit{Writers}(\overline{V}) \times \nu_w \longrightarrow \mathcal{L}$$

Assuming that access control is enforced on all XML output streams produced, we can amend the original definition of secure information flow to check access control policies specified by the security oracle Ω_w.

Definition 4.1.2 (Secure information flow)**.** *A method has secure information flow according to the Definition 6.1 of [21] with the additional constraint related to XML output streams:*

$$\forall o \in \mathit{Writers}(\overline{V})\, \forall v \in \nu_w \quad \lambda_w(\overline{o}, v) \sqsubseteq \Omega_w(\overline{o}, v)$$

Security oracle for data read from XML streams

XML documents parsed have access control policies that need to be integrated into the information flow policy. Basically, data read from XML streams have security levels that need to be mapped to their corresponding abstract values generated during AMG's computation and positioned in XML streams thanks to AXCG. The mapping of access control mechanisms for XML to information flow is concretely ensured by the oracle function

$$\Omega_r : \mathit{Readers}(\overline{V}) \times \{\omega_i.\overline{o}_j \mid \exists(\texttt{Attr}, (\omega_i, j)) \in \nu_r \vee \exists(\texttt{Text}, (\omega_i, j)) \in \nu_r\} \longrightarrow \mathcal{L}$$

Properly speaking, the assignment of security levels to abstract values corresponding to XML data read (attribute and content values) requires to slightly modify the original security function λ_v. Because of approximations in the static analysis, especially because of the allocation site model, different data coming from an XML input stream can have been abstracted with the same abstract value. For instance this case occurs if the same code (same invocation instruction in the bytecode) is used to acquire the value of attributes with the same name but attached to different opening tags. In this case it is mandatory to compute the least upper bound of corresponding security levels given by access control policies.

Definition 4.1.3 (Security function λ_v)**.** *The function λ_v from Definition 6.1 of [21] is modified to include security levels of data incoming from XML streams:*

$$\lambda_v : \overline{V} \longrightarrow \mathcal{L}$$

$$\overline{v} \longmapsto \begin{cases} \bigsqcup_{(.,.,\omega_i.\overline{o}_j)\in\xi} \Omega_r(\omega_i.\overline{o}_j, \overline{v}) & \text{if } \overline{v} = \omega_k.\overline{o}_l \text{ and } \exists(t,(\omega_k,l)) \in \nu_r \text{ and } t \in \{\texttt{Attr}, \texttt{Text}\} \\ \bigsqcup_{(\overline{v}',\overline{v},\langle f,x\rangle)\in\overline{E}} \lambda_t(f, Type(\overline{v}')) & \text{otherwise.} \end{cases}$$

Since the security function now includes access control policies for XML data, the definition of secure information flow is untouched.

4.1.2 Access control policy of an XML input/output stream

The first argument of the security oracle functions is an abstract instance of `XmlPullParser` for Ω_r, and an abstract instance of `XmlSerializer` for the function Ω_w. Outside the static analysis context, it is not convenient to deal with such entities, especially because it depends on the positions of their allocation in bytecode instructions.

As mentioned in the previous section, all access control policies applied to the XML input/output streams existing in analysed methods must share the same security lattice. Upon this lattice, each policy defines its own set of rules. To simplify the assignment of an access control policy to an XML stream, we rely on the namespace feature of the XML language.

Each access control policy is assigned a unique namespace, and each XML input/output stream must refer to elements within a unique namespace. Since access control policies are strongly related to the structure of XML documents on which they are applied, such constraints can be easily achieved in practice. Once each access control policy refers to a unique namespace, we can merge all access control policies into a unique one. In addition, a default access control policy can be appended for elements without a namespace or not matching any available access control policy.

4.1.3 Matching XPath descriptors

Most access control mechanisms for XML rely on XPath descriptors to identify parts of XML documents. We provide an algorithm to identify the set of AXCG's vertices matching a XPath descriptor.

XPath restrictions

All features of the XPath language cannot be supported because of approximations in the abstract analysis and unknown run-time control flow paths. First, we only consider absolute path descriptors to avoid cross-documents requests, *i.e.* path descriptors only starting with "/". Then, because of the loss of elements ordering, *i.e.* order between tags with the same parent tag, we are not able to use the following navigation and position-dependent operations and tests:

- `preceding`: selects the elements that are descendants of the root of the tree in which the context element is found, are not ancestors of the context element, and occur before the context element in document order;

- `following`: selects the elements that are descendants of the root of the tree in which the context element is found, are not descendants of the context element, and occur after the context element in document order;
- `preceding-sibling`: selects the children of the context element's parent that occur before the context element in document order;
- `following-sibling`: selects the children of the context element's parent that occur after the context element in document order;
- `fn:last()`: denotes the position of the last element in the context;
- `fn:position()`: denotes the position of an element in its context;
- `[`x `=` y`]`: tests the equality between x and y where x is not an attribute name (`@something`), or y is not a string literal.

From XPath to AXCG vertices

The Algorithm 4.1.1 gives the $ResolveXPath$ procedure to retrieve the set of AXCG vertices matching a given XPath descriptor according to the restrictions from the previous section. This generic algorithm relies on the procedure $ResolveElement$, for which a simplified version is given by the Algorithm 4.1.2 to illustrate how it can be implemented.

Require: An AXCG $G_X = (\nu, \xi, \eta)$ build upon an AMG $\overline{G} = (\overline{V}, \overline{E})$
Require: A XPath descriptor $P \in Strings$ where $P[i]$ denotes the $(i-1)^{\text{th}}$ substring of P delimited by "/", with $0 \leq i < n$ and n is the number of such substrings in P
Require: An abstract XML input or output stream $\overline{o} \in Writers(\overline{V}) \cup Readers(\overline{V})$

$R \leftarrow ResolveElement(G_X, P[0])$

for $i \leftarrow 1$ **to** $n - 1$ **do**
  $R \leftarrow (R \times ResolveElement(G_X, P[i]) \times \{\overline{o}\}) \cap \xi$
end for

return R

Algorithm 4.1.1: $ResolveXPath$ procedure to retrieve AXCG vertices matching a XPath descriptor.

4.2 Example of access control mechanisms for XML

The generic approach of security oracles permits to connect to many different access control mechanisms for XML. We choose the one from Gabillon *et al.* [19] to illustrate how this can be achieved in practice.

Gabillon *et al.* define a simple RBAC mechanism (Section 1.2.1) relying on two structures: the *XML Subject Sheet (XSS)* that defines roles considered in the system, and the

Require: An AXCG $G_X = (\nu, \xi, \eta)$
Require: A string $e \in Strings$ representing an XPath element

if e equals to "" **then**
  return ν
else
  if e equals to "text()" **then**
    return $\{(\texttt{Text}, (\omega_i, j)) \mid (\texttt{Text}, (\omega_i, j)) \in \nu\}$
  else
    if e starts with "@" **then**
      $a \leftarrow$ attribute name of e
      return $\{x \mid x = (\texttt{Attr}, (\omega_i, j)), x \in \nu, (\overline{N}, .) = \eta(\overline{o}, x), a \in \mathsf{qnames}(\overline{N})\}$
    else
      return $\{x \mid x \in \nu, (\overline{N}, .) = \eta(\overline{o}, x), e \in \mathsf{qnames}(\overline{N})\}$
    end if
  end if
end if

Algorithm 4.1.2: Simplified *ResolveElement* procedure to select vertices of an AXCG given a XPath element.

XML Authorisation Sheet (XAS) that defines access control rules. In order to connect to our information flow model to this access control mechanism, we first define the common lattice of security levels extracted from a single XSS. Then, we build the security oracles from a set of XAS.

4.2.1 Lattice of security levels

The *XML Subject Sheet (XSS)* defines a set of $Users$ and a set of $Groups$. A group is simply a set of users and/or other groups. Equipped with a greater element $\top$ and a lower one $\bot$, $Users \cup Groups$ is a lattice with the following ordering relation $\sqsubseteq$:

$$\forall u \in Users \bot \sqsubseteq u \sqsubseteq \top \qquad \forall g \in Groups \bot \sqsubseteq g \sqsubseteq \top \qquad \forall g \in Groups \forall x \in g \quad g \sqsubseteq x$$

4.2.2 Security oracles build from access control policies

The *XML Authorisation Sheet (XAS)* defines a set $Rules$ of access control rules. Each rule is a quadruple (s, o, a, p) where $s \in \wp(Users \cup Groups)$ is the set of users aimed by the rule, $o \in Strings$ is a XPath descriptor specifying the part(s) of the XML document aimed by the rule, $a \in \{\texttt{grant}, \texttt{deny}\}$ is the privilege given to the users in s for the described part(s), and $p \in \mathbb{Z}$ is the priority level of this rule. In this model, when the descriptor o matches a tag, it implicitly means that the corresponding rule applies to all its content and attributes, but also to all its child tags recursively.

A default privilege is set for parts of XML documents not matching any rule, which is equivalent to add a rule that matches all users, applies to the root of the document (`"/"`) and has the lowest priority. When the default privilege is set to `open` it is equivalent to set

the `grant` privilege for this rule, while the default privilege `closed` is equivalent to set the `deny` privilege.

When several rules match the same XML portion with a conflict for a user about the privilege to be given (`grant` or `deny`), the rules with the highest priority are selected. If several rules have the highest priority value, then the rule that appears last in the XAS is selected.

According to the remarks of Section 4.1.3, several constraints have to applied on access control rules defined in XAS in order to obtain a coherent system. All XPath elements occurring within each XAS must be prefixed with a common unique namespace. In addition, for simplicity, we bypass the default privilege of each XAS and define two default privileges: one for XML content read, the other one for XML content written. Given theses restrictions and hypothesis, the Algorithm 4.2.2 sketches the algorithms of Gabillon *et al.* to build the two corresponding security oracles Ω_r and Ω_w.

Require: An AXCG $G_X = (\nu, \xi, \eta)$ build upon an AMG $\overline{G} = (\overline{V}, \overline{E})$
Require: An abstract XML input or output stream $\overline{o} \in \mathit{Writers}(\overline{V}) \cup \mathit{Readers}(\overline{V})$
Require: A set of vertices $I \subseteq \nu$
  $N \leftarrow \{v_2 \mid (v_1, v_2, \overline{o}) \in \xi, v_1 \in I, v_2 \notin I\}$
  if $N = \emptyset$ **then**
    return I
  end if
  return $\mathit{Closure}(G_X, \overline{G}, \overline{o}, I \cup N)$

Algorithm 4.2.1: *Closure* procedure that computes the transitive vertex closure of a set of vertices in a given AXCG.

Require: An AXCG $G_X = (\nu_r \cup \nu_w, \xi, \eta)$ build upon an AMG $\overline{G} = (\overline{V}, \overline{E})$
Require: A set of access policy rules $Rules \in \wp(Users \cup Groups) \times Strings \times \{\texttt{grant}, \texttt{deny}\} \times \mathbb{Z}$
Require: A security lattice $(Users \cup Groups \cup \{\top, \bot\}, \sqcup, \sqsubseteq)$
Require: A default privilege $d_r \in \{\texttt{open}, \texttt{closed}\}$ common to all XML input streams
Require: A default privilege $d_w \in \{\texttt{open}, \texttt{closed}\}$ common to all XML output streams

/** Security oracle for output streams **/
$AttributesAndContents \leftarrow \{(t, (\omega_i, j)) \mid (t, (\omega_i, j)) \in \nu_r \wedge (t = \texttt{Attr} \vee t = \texttt{Text})\}$
for all $\overline{o} \in Writers(\overline{V})$ **do**
  for all $v \in \nu_w$ **do**
    $\Omega_w(\overline{o}, v) \leftarrow \begin{cases} \bot & \text{if } d_w = \texttt{open}, \\ \top & \text{if } d_w = \texttt{closed}. \end{cases}$
    $seen(v) \leftarrow \bot$
  end for

  for all $(S, x, a, p) \in Rules$ **do**
    for all $v \in Closure(G_X, \overline{G}, \overline{o}, ResolveXPath(G_X, \overline{G}, x, \overline{o}))$ **do**
      if $seen(v) = \bot$ **or** $seen(v) = p' \wedge p' \leq p$ **then**
        $\Omega_w(\overline{o}, v) \leftarrow \begin{cases} \bigsqcup_{s \in S} s & \text{if } a = \texttt{grant}, \\ \bigsqcup_{s \in ((Users \cup Groups) \setminus S)} s & \text{if } a = \texttt{deny}. \end{cases}$
        $seen(v) \leftarrow p$
      end if
    end for
  end for
end for

/** Security oracle for input streams **/
for all $\overline{o} \in Readers(\overline{V})$ **do**
  for all $v \in AttributesAndContents$ **do**
    $\Omega_r(\overline{o}, v) \leftarrow \begin{cases} \bot & \text{if } d_r = \texttt{open}, \\ \top & \text{if } d_r = \texttt{closed}. \end{cases}$
    $seen(v) \leftarrow \bot$
  end for

  for all $(S, x, a, p) \in Rules$ **do**
    for all $v \in Closure(G_X, \overline{G}, \overline{o}, ResolveXPath(G_X, \overline{G}, x, \overline{o})) \cap AttributesAndContents$ **do**
      if $seen(v) = \bot$ **or** $seen(v) = p' \wedge p' \leq p$ **then**
        $\Omega_r(\overline{o}, v) \leftarrow \begin{cases} \bigsqcup_{s \in S} s & \text{if } a = \texttt{grant}, \\ \bigsqcup_{s \in ((Users \cup Groups) \setminus S)} s & \text{if } a = \texttt{deny}. \end{cases}$
        $seen(v) \leftarrow p$
      end if
    end for
  end for
end for

Algorithm 4.2.2: Building the security oracles Ω_w and Ω_r for Gabillon *et al.*.

Conclusion

This report introduces foundations to properly include data read from and written to XML documents with access control policies in a static information flow analysis for Java bytecode. We first investigate in Chapter 2 the different means to read/write XML content in Java, and exhibit their (dis)advantages in constrained environments and for the purpose of a static analysis. In Chapter 3 we describe the *Abstract XML Content Graph (AXCG)*, a graph structure to track data flowing from and to XML streams through the XMLPull API, and the modifications brought to our original information flow model to include AXCG's building. We finally detail in Chapter 4 security oracles, two functions that ensure the connection between a secure information flow policy applying to the set of analysed classes and access control policies attached to XML input and output streams.

The work presented in this report builds on an existing information flow analysis for Java bytecode. The major advantage of the original model has been preserved: secure information flow policy, and now access control policies of XML streams, do not require to be known during the analysis. Changes to the whole security policy can be made *a posteriori* without the need to re-conduct the analysis. The second advantage of the solution presented is its genericity: any access control mechanism for XML that follows the RBAC approach can be used to define the security policies of XML input/output streams.

Since we conduct an abstract interpretation in a static way, the structure of some generated/parsed XML documents cannot necessarily be inferred correctly. This is especially the case when implementations involve intricate control flow paths. It is conceivable to propose *dynamic* artifacts, such as code injection or run-time monitoring, to achieve a more precise verification, but such mechanism are not suitable for small constrained systems.

In order to fully include this work in the original information flow framework, it is required to get into practical details such as how to extend the proof-elements embedded in the bytecode for the PCC-like on-device verification of bytecode at loading-time. As is, AXCGs are actually too large to be embedded as approximated AMGs are with STAN tool [20]. A practical solution has to be found to loosen AXCGs with a good compromise between kept information and memory footprint.

A second perspective of this work is to provide a verification of generated/parsed XML documents against an expected XML structure described, for instance, by a DTD or XML Schema sheet. Conversely, it could also be feasible to statically infer valid structure descriptors for generated/parsed XML documents.

Finally it could be interesting not to deal only with access control policies of XML data but also with policies on structural elements such as the presence/absence of a particular tag or attribute.

Bibliography

[1] XML specifications.

[2] XPath 1.0 specifications.

[3] XPath 2.0 specifications.

[4] Belokosztolszki, A., Eyers, D. M., and Moody, K. Policy contexts: controlling information flow in parameterised rbac. In *POLICY03* (Lake Como, Italy, June 2003), IEEE Computer Society, pp. 99–110.

[5] Benzaken, V., Burelle, M., and Castagna, G. Information flow security for XML transformations. In *ASIAN03* (Mumbai, India, December 2003), V. A. Saraswat, Ed., vol. 2896, Springer, pp. 33–53.

[6] Benzaken, V., Castagna, G., and Frisch, A. CDuce: an XML-centric general-purpose language. *SIGPLAN Notices 38*, 9 (2003), 51–63.

[7] Bertino, E., and Ferrari, E. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security 5*, 3 (2002), 290–331.

[8] Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P. Design and implementation of an access control processor for XML documents. *Computer Networks 33*, 1–6 (June 2000), 59–75.

[9] Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P. Securing XML documents. In *EDBT00* (Konstanz, Germany, March 2000), C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, Eds., vol. 1777 of *Lecture Notes in Computer Science*, Springer, pp. 121–135.

[10] Damiani, E., Fansi, M., Gabillon, A., and Marrara, S. A general approach to securely querying XML. *Computer Standards & Interfaces 30*, 6 (2008), 379–389.

[11] Dukovich, A., Hua, J., Lee, J. S., Huffman, M., and Dekhtyar, A. JOXM: Java object XML mapping. In *ICWE08* (New York, USA, July 2008), D. Schwabe, F. Curbera, and P. Dantzig, Eds., IEEE Computer Society, pp. 332–335.

[12] Estlund, M. J. A survey and analysis of access control architectures for XML data. Master's thesis, Naval Postgraduate School, Monterey, California, USA, March 2006.

[13] Fan, W., Chan, C. Y., and Garofalakis, M. N. Secure XML querying with security views. In *SIGMOD04* (Paris, France, June 2004), G. Weikum, A. C. König, and S. Deßloch, Eds., ACM, pp. 587–598.

[14] Fernández, E. B., Gudes, E., and Song, H. A model for evaluation and administration of security in object-oriented databases. *IEEE Transactions on Knowledge and Data Engineering 6*, 2 (April 1994), 275–292.

[15] Ferraiolo, D., and Kuhn, R. Role-based access controls. In *15th National Computer Security Conference* (October 1992), pp. 554–563.

[16] Fontaine, A., Ghindici, D., Hym, S., Simplot-Ryl, I., and Talbot, J.-M. D3 - expression of security policies compatible with code sharing. Tech. rep., ANR SFINCS (LIFL, Verimag), 2010.

[17] Foster, N. J., Pierce, B. C., and Zdancewic, S. Updatable security views. In *CSF09* (Port Jefferson, New York, USA, July 2009), IEEE Computer Society, pp. 60–74.

[18] Gabillon, A. A logical formalization of a secure XML database. *Computer Science and System Engineering Journal 21*, 5 (September 2006).

[19] Gabillon, A., and Bruno, E. Regulating access to XML documents. In *DBSEC01* (Ontario, Canada, July 2001), M. S. Olivier and D. L. Spooner, Eds., vol. 215, Kluwer, pp. 299–314.

[20] Ghindici, D., and Simplot-Ryl, I. On practical information flow policies for Java-enabled multiapplication smart cards. In *Proc. 8th IFIP Conf. on Smart Card Research and Advanced Applications (CARDIS08)* (Egham, Surrey, UK, 2008), vol. 5189, pp. 32–47.

[21] Ghindici, D., Simplot-Ryl, I., and Talbot, J.-M. A sound analysis for secure information flow using abstract memory graph. In *FSEN09* (Kish Island, Iran, 2010), F. Arbab and M. Sirjani, Eds., vol. 5961 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 355–370.

[22] Groz, B., Staworko, S., Caron, A.-C., Roos, Y., and Tison, S. XML security views revisited. In *DBPL* (2009), P. Gardner and F. Geerts, Eds., vol. 5708 of *Lecture Notes in Computer Science*, Springer, pp. 52–67.

[23] Hada, S., and Kudo, M. XML access control language: provisional authorization for XML documents.

[24] Kanza, Y., Mendelzon, A. O., Miller, R. J., and 0002, Z. Z. Authorization-transparent access control for XML under the non-truman model. In *EDBT* (2006), Y. E. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Böhm, A. Kemper, T. Grust, and C. Böhm, Eds., vol. 3896 of *Lecture Notes in Computer Science*, Springer, pp. 222–239.

[25] Kuper, G. M., Massacci, F., and Rassadko, N. Generalized XML security views. *International Journal of Information Security 8*, 3 (2009), 173–203.

[26] Pallis, G., Stoupa, K., and Vakali, A. Storage and access control issues for XML documents. In *Encyclopedia of Information Science and Technology (V)*, M. Khosrow-Pour, Ed. Idea Group, 2005.

[27] Papakonstantinou, Y., and Vianu, V. DTD inference for views of XML data. In *PODS* (2000), ACM, pp. 35–46.

[28] Rabitti, F., Bertino, E., Kim, W., and Woelk, D. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems 16*, 1 (March 1991), 88–131.

[29] Ray, I., and Muller, M. Using schemas to simplify access control for XML documents. In *ICDCIT04* (Bhubaneswar, India, December 2004), R. K. Ghosh and H. Mohanty, Eds., vol. 3347 of *Lecture Notes in Computer Science*, Springer, pp. 363–368.

[30] Rizvi, S., Mendelzon, A. O., Sudarshan, S., and Roy, P. Extending query rewriting techniques for fine-grained access control. In *SIGMOD Conference* (2004), G. Weikum, A. C. König, and S. Deßloch, Eds., ACM, pp. 551–562.

[31] Samarati, P., Bertino, E., and Jajodia, S. An authorization model for a distributed hypertext system. *IEEE Transactions on Knowledge and Data Engineering 8*, 4 (August 1996), 555–562.

[32] Seehusen, F., and Stølen, K. Information flow property preserving transformation of uml interaction diagrams. In *SACMAT* (Lake Tahoe, California, USA, June 2006), D. Ferraiolo and I. Ray, Eds., ACM, pp. 150–159.

[33] Wong, E. Y. C., Chan, A. T. S., and Leong, H.-V. Semantic-based approach to streaming XML contents using Xstream. In *COMPSAC03* (Dallas, Texas, USA, November 2003), IEEE Computer Society, pp. 91–.

[34] Zhang, X., Park, J., and Sandhu, R. S. Schema based XML security: RBAC approach. In *DBSEC03* (Estes Park, Colorado, USA, August 2003), S. De Capitani di Vimercati, I. Ray, and I. Ray, Eds., Kluwer, pp. 330–343.

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803