Static Analysis for Event-Based XML Processing

Anders Møller

Department of Computer Science
University of Aarhus
amoeller@brics.dk

Abstract

Event-based processing of XML data – as exemplified by the popular SAX framework – is a powerful alternative to using W3C's DOM or similar tree-based APIs. The event-based approach is particularly superior when processing large XML documents in a streaming fashion with minimal memory consumption.

This paper discusses challenges for creating program analyses for SAX applications. In particular, we consider the problem of statically guaranteeing that a given SAX program always produces only well-formed and valid XML output. We propose an analysis technique based on existing analyses of Servlets, string operations, and XML graphs.

1. Introduction

Most existing work on providing static guarantees about programs that manipulate XML documents has concentrated on programming languages or APIs that assume a tree-view of XML documents. (A survey is presented in [19].) Naturally, this takes on a high-level view of XML that implies convenient programming models and permits sophisticated type systems or program analyses. However, many real-world applications are built using a fundamentally different model where XML documents are viewed as streams of events, as produced by an XML parser encountering tags and character data while reading documents left-to-right. For many applications this model leads to significantly lower memory consumption although it is often regarded more difficult to program with. The most well-known event-based framework is SAX [1], which is based on Java.

The goal is to provide static analysis for SAX, as a step towards complementing the existing work for tree-based XML transformation systems. Specifically, we attack the following problems for a given SAX application:

- if the application produces XML output, is the output guaranteed to be well-formed and valid (according to some given schema)?
- if the input is XML and we have a schema describing the possible input, how can that schema be exploited to improve precision of analyses of the Java code?
- if both input and output are XML, does validity of the input imply validity of the output?

To the extent possible, we wish to solve these problems without changing the SAX framework. In particular, we avoid imposing significant restrictions on the programming style or requiring schema-based type annotations.

The approach suggested here builds upon existing work on XML graphs [20], static analysis for the XACT system [14, 12], analysis for Java Servlets [13], and analysis of string operations in Java [4]. It would of course be an interesting challenge to instead try building on alternative techniques, for example regular expression types [9], but we leave that to others.

Although SAX is widely used, it is often considered a low-level tool for stream processing XML data. An entirely different approach is to develop high-level languages that are compiled into event-based programs and then obtain static guarantees using type checking on the source code. For example, stream processors can be derived from macro forest transducers [21], and TransformX is a related approach based on extended regular tree grammars [22]. Other related work targets stream processing for XPath [8] or XQuery [6] and development of new high-level languages, such as STX [5] and XStream [7], but usually without considering validity guarantees. Another use of static analysis is to check that a program written in a tree-oriented XML transformation language processes its input in a linear way that permits streaming [16]. The results presented here complement the existing work by demonstrating that static validity analysis is realistic for event-based XML processing with general purpose languages.

In addition to focusing on the particular problem of analyzing SAX applications, the issues being raised here in many cases have a more general nature where solutions may also be useful for other program analyses that work on Java code, for example the need for precise modeling of field variables and conditional statements. Conversely, it should ideally be possible to develop complex specialized program analyses, such as this one for SAX, in a compositional manner from simpler analyses that each focus on one particular aspect. Furthermore, the considerations presented here may (together with our analysis for XSLT [18]) provide inspiration for developing type checking or static analysis for the domain-specific event-based XML transformation language STX.

Overview

First, in Section 2, we give a brief introduction to the SAX 2.0 framework, discuss some of the challenges it imposes on static analysis, and present some simple but typical example programs. Section 3 outlines a well-formedness and validity analysis of Java programs that produce SAX events as output, and Section 4 considers the more difficult problem of reasoning about SAX filters and relating input schemas with the control-flow and data-flow in the program. Section 5 concludes by summarizing the key ideas and contributions and suggesting future work.

2. The SAX 2.0 Framework and Challenges for Static Analysis

With SAX 2.0, an XML document is viewed as a stream of events, the most important being of the following kinds: *start document*, *end document*, *start element*, *end element*, and *characters*. The most central constituent is the `ContentHandler` interface, which contains a method for each kind of event. In particular, the method `startElement` has arguments for the element name, its namespace URI, and the attributes. The latter are represented via the interface `Attributes`, which allows access to attributes either as an ordered list or as a name–value map. Namespace declarations are represented by two additional event handler methods: `startPrefixMapping`, which has arguments for the prefix and the URI of a namespace declaration, and `endPrefixMapping`, which marks the end of the scope of a namespace declaration.

`XMLReader` is an interface for parsers that produce events from, for example, the textual representation of XML documents. The `XMLWriter` class is a simple example of an implementation of the `ContentHandler` interface that converts in the other direction: from events to (hopefully well-formed and namespace compliant) textual XML documents.

A common way to implement `ContentHandler` classes is to extend the class `DefaultHandler`, which provides empty event handlers for all kinds of events.

EXAMPLE 1. *Assume that we want to convert a collection of* `Card` *objects, described by the following class, into an XML stream representation.*

```
class Card {
  int id;
  String name;
  List<String> emails;
  String phone; // null represents "not available"
}
```

The XML representation is described by the following schema, `cards.xsd` *(using XML Schema notation):*

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org"
        elementFormDefault="qualified">

  <element name="cards">
    <complexType>
      <sequence>
        <element name="card"
                 minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="name" type="string"/>
              <element name="email" type="string"
                       minOccurs="0"
                       maxOccurs="unbounded"/>
              <element name="phone" type="string"
                       minOccurs="0"/>
            </sequence>
            <attribute name="id" type="integer"/>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>

</schema>
```

The output could be the following sequence of events for a single `Card` *object:*

start document
start prefix mapping "" $\mapsto$ `http://businesscard.org`
start element `cards`
start element `card` *with attribute* `id=42`
start element `name`
characters `John Doe`
end element `name`
start element `email`
characters `john.doewidget.inc`
end element `email`
end element `card`
end element `cards`
end prefix mapping ""
end document

This conversion can be achieved with the following method that generates the appropriate SAX events to a `ContentHandler` *(which may be an instance of* `XMLWriter`*):*

```
void cards2xml(Collection<Card> cards, ContentHandler out)
    throws SAXException {
  String NS = "http://businesscard.org";
  out.startDocument();
  out.startPrefixMapping("", NS);
  out.startElement(NS, "cards", null,
                   new AttributesImpl());
  for (Card c : cards) {
    AttributesImpl empty_attr = new AttributesImpl();
    AttributesImpl attr = new AttributesImpl();
    attr.addAttribute("", "id", null, null,
                      Integer.toString(c.id));
    out.startElement(NS, "card", null, attr);
    out.startElement(NS, "name", null, empty_attr);
    out.characters(c.name.toCharArray(),
                   0, c.name.length());
    out.endElement(NS, "name", null);
    for (String email : c.emails) {
      out.startElement(NS, "email", null, empty_attr);
      out.characters(email.toCharArray(),
                     0, email.length());
      out.endElement(NS, "email", null);
    }
    if (c.phone != null) {
      out.startElement(NS, "phone", null, empty_attr);
      out.characters(c.phone.toCharArray(),
                     0, c.phone.length());
      out.endElement(NS, "phone", null);
    }
    out.endElement(NS, "card", null);
  }
  out.endElement(NS, "cards", null);
  out.endPrefixMapping("");
  out.endDocument();
}
```

(We here ignore the QName arguments to `startElement`/`endElement` *and the type argument to* `addAttribute`*.) The big question is: is the output always well-formed, namespace compliant, and valid relative to the schema? (Unlike the tree-based XML processing frameworks, not even well-formedness and namespace compliance are guaranteed here, but in return it consumes a minimal amount of memory.) Our program analysis should be able to automatically verify whether this is indeed the case.*

This tiny example already exposes a number of non-trivial challenges for static analysis:

(1) The analysis must be able to extract an approximation of the possible sequences of events and transform it into a representation that is amendable to checking well-formedness and validity, preferably with the widely used schema language XML Schema.

(2) Element names, attribute names, attribute values, namespaces URIs, and character data all come from, in general, dynamically computed strings, so the analysis must be capable of reasoning about string operations in general Java code.

(3) The argument to the `characters` method is a substring that is given as an interval of a character array – so to obtain good analysis precision, the character array and the interval bounds must be tracked collectively by the analysis.

(4) The `Attributes` interface and its implementing classes must obtain special treatment to be able to reason about attributes in the resulting XML documents.

A SAX *filter*, represented by the interface `XMLFilter`, is a specialization of `XMLReader` that obtains its events from another XML reader rather than a primary source like a textual XML document. Thus, a filter is an XML transformation that takes events as input and produces events as output. Typically, filters are implemented as subclasses of `XMLFilterImpl`, which is both an `XMLFilter` and a `ContentHandler`, by itself acting as the identity transformation. In subclasses of `XMLFilterImpl`, events can be modified by overriding the event handler methods and producing events by invoking the appropriate event handler methods in the super class. This design allows filters to be pipelined to make composite XML transformations.

EXAMPLE 2. *The following filter takes as input a document that is valid according to* `cards.xsd` *(such as, the output from Example 1) and produces as output a list of the* `name` *elements that appear inside* `card` *elements where a* `phone` *element is present:*

```
class NamesFilter extends XMLFilterImpl {
  private static final String NS =
    "http://businesscard.org";

  private boolean is_name, has_phone;
  private StringBuffer name;

  public NamesFilter() {}

  public NamesFilter(XMLReader parent) {
    super(parent);
  }

  public void startElement(String uri, String localName,
                           String qName, Attributes atts)
      throws SAXException {
    is_name = localName.equals("name");
    if (is_name)
      name = new StringBuffer();
    if (localName.equals("phone"))
      has_phone = true;
  }

  public void characters(char[] ch, int start,
                         int length)
      throws SAXException {
    if (is_name)
      name.append(ch, start, length);
  }

  public void endElement(String uri, String localName,
                         String qName)
      throws SAXException {
    if (localName.equals("card") && has_phone) {
      AttributesImpl empty_attr = new AttributesImpl();
      super.startElement(NS, "name", null, empty_attr);
      super.characters(name.toString().toCharArray(),
                       0, name.length());
      super.endElement(NS, "name", null);
      has_phone = false;
    }
  }
}
```

When executed, this program simultaneously consumes and produces XML events. (One may argue that programs like this are difficult to understand, but SAX is nevertheless being used extensively in practice.)

The filter uses several recurring pattern in SAX programs: First, field variables are used to correlate events. In this particular program, when a start tag is encountered, information about its name is stored to be able to determine whether or not character data is relevant and output should be emitted. Second, the SAX specification allows contiguous character data to be reported as several consecutive `characters` *events, so the name data is collected via a* `StringBuffer`.

The following simple XML Schema type describes the intended output format:

```
<complexType name="Names">
  <sequence minOccurs="0" maxOccurs="unbounded">
    <element name="name" type="string"/>
  </sequence>
</complexType>
```

The big question is now: given that the input to the filter is valid according to `cards.xsd`, *is the output always valid according to the* `Names` *type?*

This example illustrates a number of additional challenges for the static analysis:

(5) The control-flow and data-flow clearly depends on the possible sequences of input events, so in order to reason precisely about flow, the input schema must be taken into account. In particular, the following problem must be addressed: Given a schema (written in XML Schema), we need a representation of the event sequences that correspond to valid documents, in a way that can be combined with control/data-flow graphs. Moreover, we need to consider the possibility of pipelining filters. Do we need schema annotations as pre/post conditions at each filter, or is it possible to reason fully automatically about a whole pipeline? Fortunately, it appears that filter pipelines are usually fixed at compile time rather than being assembled dynamically.

(6) Field variables in the filter object are commonly used for transferring information between event handler methods, such as the two boolean flags and the string buffer in the example above. The analysis must be able to model such fields flow sensitively and with strong updating to maintain precision. (More generally, one could imagine SAX programs that materialize short event sequences into DOM trees, but that does not appear to be common in practice.)

(7) The analysis must be path sensitive to properly model the effect of the conditional statements in the event handlers. At least, it cannot disregard boolean operations and simple string comparisons.

EXAMPLE 3. *The examples shown above are closely connected to concrete XML languages described by schemas. However, some SAX filters are more generic. As an example, the following filter runs on input from any XML language and, allegedly, strips all SAX events related to "Bookmarks channels":*

```
public class BookmarksRemover extends XMLFilterImpl {
  private boolean stripIt = false;

  public void startElement(String uri, String localName,
                           String qName, Attributes atts)
      throws SAXException {
    if (qName.equals("channel")) {
      String name = atts.getValue("name");
      if (name != null && name.equals("Bookmarks"))
        stripIt = true;
    }
    if (!stripIt)
      super.startElement(uri, localName, qName, atts);
  }
```

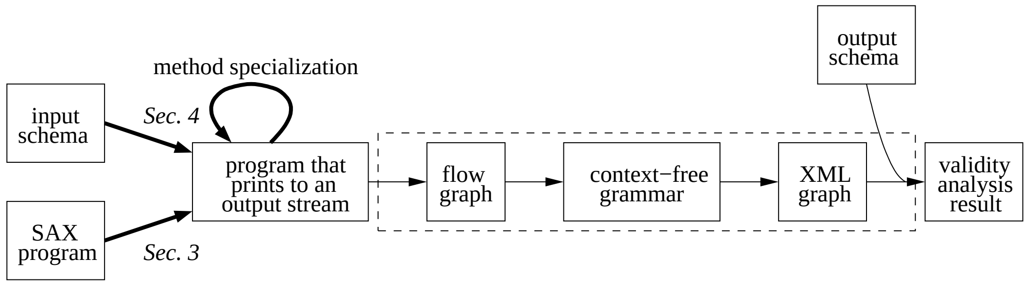


Figure 1. Structure of the analysis. The dashed box contains parts that are described in earlier papers [13, 12].

```
  public void endElement(String uri, String localName,
                         String qName)
      throws SAXException {
    if (!stripIt)
      super.endElement(uri, localName, qName);
    if (stripIt && qName.equals("channel"))
      stripIt = false;
  }

  public void characters(char[] ch, int start,
                         int length)
      throws SAXException {
    if (!stripIt)
      super.characters(ch, start, length);
  }
}
```

(This is a condensed version of some program code found at `uportal.org`*.) In SAX programs like this where schemas are not available, our program analysis will focus on well-formedness, in particular on balancing of start and end tags in the output. In this case, however, well-formedness may fail for some input, which we return to in Section 4.*

It should be evident from the discussions above that developing a high-precision static analyzer for SAX applications is a considerable task with plenty of obstacles. Nevertheless, the situation is far from hopeless: First, SAX applications tend to be fairly small, at least if slicing away code that is not directly related to producing or consuming events. Second, as argued in the following, many of the challenges seem closely related to problems that have been attacked by other program analysis techniques in the past, and others can be viewed as inspiration for developing new analysis techniques for general Java programs.

3. Analyzing SAX Event Producers

As a modest first step, we will focus on programs that only *produce* SAX events, like Example 1.

This problem is remarkably close to analyzing the output of applications built with Java Servlets, which is the topic of the paper [13] and described briefly below. (See also the related work by Minamide and Tozawa [17].) With servlets, output is generated by printing strings to an output stream in a way that hopefully results in well-formed and valid XML documents. To reason about generation of SAX events instead, the idea is simply to treat event generation, for example `endElement(..., "E", ...)` (let us for now ignore namespaces), as an alternative way of writing the string `</E>` to a servlet-like output stream. If we slightly rewrite Example 1 in this way by outputting strings to a `PrintWriter` stream rather than outputting events to a `ContentHandler`, the connection to servlet analysis becomes obvious:

```
void cards2xml(Collection<Card> cards, PrintWriter out) {
  out.print("<cards>");
  for (Card c : cards) {
    out.print("<card id=\""+Integer.toString(c.id)+"\">");
    out.print("<name>");
    out.print(escapeXML(c.name));
    out.print("</name>");
    for (String email : c.emails) {
      out.print("<email>");
      out.print(escapeXML(email));
      out.print("</email>");
    }
    if (c.phone != null) {
      out.print("<phone>");
      out.print(escapeXML(phone));
      out.print("</phone>");
    }
    out.print("</card>");
  }
  out.print("</cards>");
}
```

(We here use a method `escapeXML` for escaping the special XML characters, `<`, `&`, etc.) In other words, the key idea is to translate the SAX event generation method invocations into servlet-like stream printing commands and then apply the existing analysis. In fact, the situation is in a way simpler than in the general servlet analysis since output here always comes in entire tags rather than in individual characters. (Technically, some of the grammar transformation steps in [13] then become superfluous.) Building on string analysis [4], the servlet analysis is already capable of reasoning about the possible values of dynamically computed strings, so we already have a grip on challenges (1) and (2) from Section 2.

Hence, our approach is to reduce the problem of analyzing SAX event producers to string analysis and servlet analysis. The following sections describe the reduction in more detail. The structure of the analysis is illustrated in Figure 1.

3.1 Analysis of String Operations

The Java String Analyzer, as presented in [4], works by first extracting an abstract flow graph from the given Java program, and then converting that into a context-free grammar (extended with a suitable collection of additional string operations). Finally, regular approximations are applied to obtain, for each program point of interest, a finite-state automaton whose language approximates the set of strings that may appear at runtime.

The existing string analysis tool has some well-known limitations regarding precision, but recent work [3, 10] shows that it can smoothly be extended with standard techniques for better heap modeling [2] and context sensitivity [23].

As an example, the string analyzer provides the information that the possible values of the expression `Integer.toString(c.id)` in Example 1 are described by the regular expression

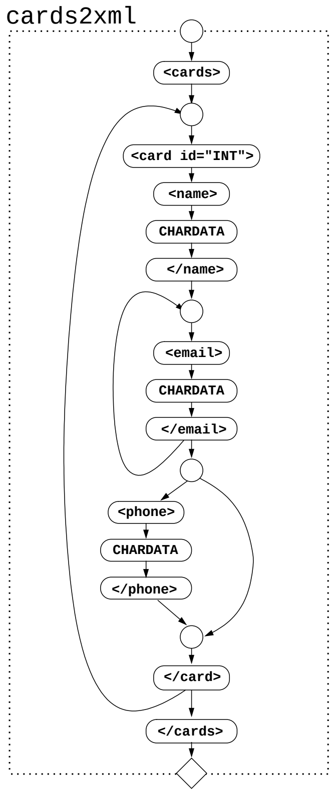


Figure 2. Flow graph for Example 1.

`0|-?[1-9][0-9]*` (or rather, an equivalent automaton), which eventually will allow the remaining analysis to verify validity of the generated `id` attributes. In fact, the analyzer will also point out that the values of the `name`, `email`, and `phone` fields can be *any* strings, including characters, such as Unicode code point 0, that are not allowed in XML documents, so the resulting output will in this case not be well-formed XML, which the subsequent output stream analysis will report. (In this particular case, the actual possible values of those strings will presumably be more well behaved. This can be controlled by the programmer using a type annotation feature in the string analyzer.)

3.2 Analysis of String-based Output Streams

The overall structure of the analysis of output streams, as presented in [13], is as follows. First, it runs the string analyzer to obtain a regular language for all string expressions that appear as arguments to operations that print to the output stream. Based on these regular languages, a *flow graph* is constructed for modeling the order of output stream operations and their arguments. The flow graph is divided into fragments corresponding to the methods in the program. Edges represent control flow, and nodes have the following kinds:

- append nodes describe operations that output strings to the stream (where the possible strings are represented by automata);
- invoke nodes describe method invocations and are labeled with the possible targets;
- return nodes describe method exits[1]; and
- nop nodes correspond to flow join points.

[1] In [13], return nodes are defined as a special kind of nop nodes.

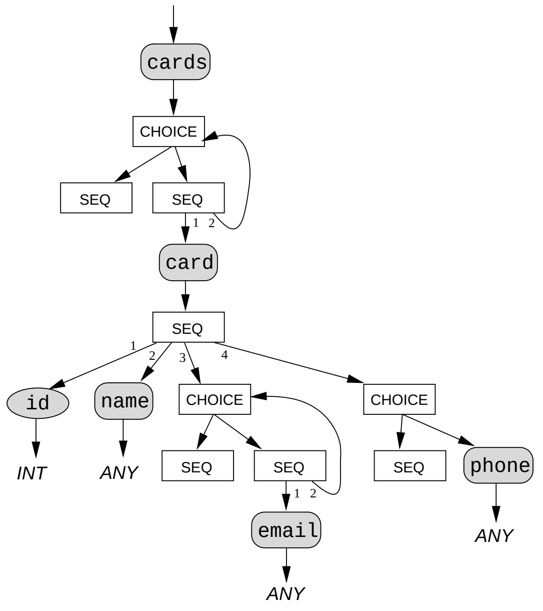


Figure 3. XML graph for Example 1.

As an example, the flow graph for the program from Example 1 is shown in Figure 2. The circles are `nop` nodes, the diamond is a `return` node, and the rounded boxes are `append` nodes where `INT` is the regular language of integer literals from Section 3.1 and `CHARDATA` represents arbitrary character data.

The flow graph is then transformed straightforwardly into a context-free grammar whose language approximates the possible output of running the program.

The next phase of the analysis checks that all strings in the language of the grammar are well-formed XML documents. This is done in three steps: First, the grammar is converted to a balanced grammar (treating the tag delimiters `<` and `</` as left and right parentheses, respectively) using a modified version of Knuth's algorithm [15]. Second, the balanced grammar is converted to a grammar on tag-form, if possible. (If not, then there are non-well-formed documents in the language.) A grammar on tag-form clearly shows the XML element tags and attributes that appear in the derivable strings. Third, it is checked that these tags and attributes satisfy the requirements for XML well-formedness (and namespace compliance).

If the well-formedness check is passed, the last phase of the analysis converts the grammar to an XML graph [20] (also called a *summary graph* in earlier papers). The XACT project [12] provides an algorithm for checking language inclusion between an XML graph and a schema written in XML Schema, which gives the final step to checking validity.

For the output stream version of Example 1, the analysis will infer the XML graph shown in Figure 3 using the graphical notation explained in [20, 13]. As this example indicates, there are different kinds of nodes in XML graphs: *element* nodes, *attribute* nodes, *text* nodes, *sequence* nodes, and *choice* nodes. (See [20, 13] for a more formal description.) The *text* node *INT* is, again, the regular language for integer literals, and *ANY* is the set of all Unicode strings (including special characters such as `<`).

3.3 Considerations Regarding Analysis Precision

The analysis is conservative: the language of the XML graph being produced contains all XML documents that may be output at runtime. Since SAX is based on a Turing complete language, spurious errors are inevitable. Naturally, the programming style influences the precision of the analysis. As an example of a potential source of imprecision, element name strings can be computed dynamically and used for producing events in the following style:

```
String x = ...;
out.startElement(..., x, ..., ...);
out.endElement(..., x, ..., ...);
```

The existing string analysis may be able to find out that the only possible values of `x` are, say, `A` and `B`, but to avoid spurious well-formedness warnings it is necessary to be able to determine and exploit the fact that `x` always has the same value when `startElement` and `endElement` are invoked. One approach to achieve this is to augment the context-free grammars being produced with knowledge about such string identities and take that into account when performing the well-formedness checking step. In case `x` has finitely many possible values, another approach is to copy the flow graph fragments to perform a polyvariant analysis of the relevant program parts.

A related hypothetical challenge is path sensitive output:

```
if (b)
  out.startElement(..., "E", ..., ...);
...
if (b)
  out.endElement(..., "E", ..., ...);
```

Arguing that the start and end tags are balanced in the output here involves reasoning about modifications of `b` between the two conditionals, which is beyond the capabilities of the current analysis.

Furthermore, heuristic method inlining or related techniques may be necessary when analyzing programs that use wrappers for the event generators, such as this one:

```
void open(String tag) throws SAXException {
  out.startElement(NS, tag, null, new AttributesImpl());
}
```

We return to some of these issue in Section 4 and leave others to future work. Results in this direction may also be useful for other program analyses that are based on XML graphs [20]. Nevertheless, a small study of existing SAX programs indicates that the basic analysis presented above often appears to be sufficient in practice, regarding validation of element structure [11].

3.4 Remaining Issues

As explained above, the two main challenges (1) and (2) from Section 2 are handled via the string analysis and the servlet analysis. An implementation based on this will be able to catch errors related to, in particular, the structure of elements in the output. However, we are in a good position to also address namespaces, character data, and attributes, despite their peculiarities in SAX.

Namespaces

Namespace events can be handled by treating `startPrefixMapping` and `endPrefixMapping` as generating special tags, such as, `<:P: ns="N">` and `</:P:>` for prefix `P` and namespace URI `N`, and then – at the level of XML graphs – transform this into ordinary namespace mappings. However, for elements with multiple namespace declarations, the SAX specification allows the invocations of `endPrefixMapping` to come in any order, so the analysis might need to reorder them to match the the invocations of `startPrefixMapping`.

Character Data

Regarding the character array intervals, i.e. challenge (3), the following observations have been made on a small collection of SAX programs [11]: First, the typical case where a string is converted into a character array interval (as in Example 1) is easily recognized and modeled with flow graphs. Second, most other character array intervals come as arguments in the `characters` event handler method in filters or content handlers, and in these cases it is unlikely that any integer operations are performed on the interval end points or that the array content is modified. This means that it appears to be sufficient to track character array intervals that flow unmodified from arguments in the `characters` event handler method to arguments in the event construction method.

Attributes

Regarding construction of objects of type `Attributes`, i.e. challenge (4), we concentrate the effort on the standard implementing class `AttributesImpl` as other implementations are uncommon. If furthermore only considering the basic methods `addAttribute` and `clear`, then these objects can be represented as finite maps from attribute names to attribute values where both are modeled as regular languages.

4. Analyzing SAX Event Consumers

To reason about applications that consume SAX events, such as content handlers or filters like Examples 2 and 3, the main problem is how to incorporate the schemas that describe the possible input (challenge (5) in Section 2).

The key idea for attacking this problem is to translate the input schema into program code that corresponds to the possible events being generated by a SAX parser reading valid input. This code then acts as a “main” method that invokes the appropriate event handlers. The combined program – consisting of this main method and the actual code to be analyzed – is then processed as described in the previous section. In other words, our strategy is to reduce the problem of reasoning about filters that both consume and produce events to reasoning about programs that only produce events.

When analyzing generic filters, such as `BookmarksResolver` from Example 3 where no schemas are available, it can still be useful to verify well-formedness of the output. A simple approach to do this is to just supply a default input schema corresponding to every possible event sequence.

4.1 Converting Schemas to Event Producer Code

The XACT project provides a translation from schemas written in XML Schema into XML graphs [12, 20]. For instance, the XML graph being generated for the schema `cards.xsd` from Section 2 is essentially the one shown in Section 3.2 (except that the `id` attributes are optional in the schema and XML Schema `xsi` attributes are permitted). Each node in the XML graph is then translated into a Java method as follows:

- An *element* node results in a method that first calls `startElement`, then it calls the method that corresponds to its content node, and finally it calls `endElement`.
- A *sequence* node, which has a sequence of successor nodes, becomes a method that calls each of the corresponding methods in order.
- A *choice* node, which has a set of successor nodes, is modeled by nondeterministically invoking the corresponding methods.

(We defer the modeling of *attribute* and *text* nodes to Section 4.4.) For the root node we add an extra method called `main`, which first calls `startDocument` then the method corresponding to its content node and finally `endDocument`.

The regular languages being used in XML graphs for describing element names, attribute names, attribute values, and character data can be represented by an operation `reg(r)` that nondeterministically returns a string from a given regular language r (although the languages are mostly singletons). Also, we allow the boolean expression `?` that nondeterministically evaluates to `true` or `false`.

The translation has the property that the set of event sequences corresponding to XML documents that are valid according to the schema coincides with the event sequences that can be generated from the Java code.

Continuing the examples, the schema `cards.xsd` is converted into the following code (again, ignoring namespaces and attributes):

```
void main() {
  startDocument();
  element1();
  endDocument();
}

void element1() {
  startElement(..., "cards", ...);
  choice1();
  endElement(..., "cards", ...);
}

void choice1() {
  if (?)
    seq1();
  else
    seq2();
}

void seq1() {
}

void seq2() {
  element2();
  choice1();
}

void element2() {
  startElement(..., "email", ..., ...);
  choice1();
  endElement(..., "email", ..., ...);
}

⋮
```

We can now combine this with the filter code, such as the `NamesFilter` class from Example 2, resulting in a program that produces – but no longer consumes – SAX events. This program can now be analyzed as in Section 3. The precision of this analysis will not be acceptable, however, since we have not yet addressed challenges (6) and (7), which are the topic of the next section.

4.2 Method Specialization

Since a filter has all start element event handling combined into one single method, `startElement` – and similarly for `endElement`, these methods need to be analyzed polyvariantly. (One might argue that an alternative design of SAX using one event handler method per element name would have been more manageable for the programmers – and it would allow us to simplify this step in the analysis.)

To model field variables, we can take advantage of a pattern that appears to be common in SAX filters: The fields that are relevant for well-formedness and validity are typically booleans that are modified only from inside the filter (often they are declared `private`), and by only one thread. This means that each field can be modeled flow sensitively as a global variable.

Inspired by Whaley and Lam [25], we propose to obtain context sensitivity by cloning methods for every context of interest and running a context insensitive analysis over the expanded program code to weed out infeasible paths, thereby specializing the relevant methods in the program:

(1) First, extract a finite collection of *element names* (or, in case of wildcards, regular sets of names), for example those being declared by the input schema, and a collection of *flag variables*, for example those that are declared as private boolean fields in the filter class.

(2) Now clone the `startElement`, `endElement`, and `characters` event handler methods in the filter class such that we have one copy for each combination of an element name and valuation of the flag variables.[2] Each invocation of one of these methods is changed into a nondeterministic choice of the new copies.

(3) Finally, we are in a position to perform interprocedural constant propagation and eliminate unreachable code (that is, conditional branches whose guards are constantly `false`) as in the *Conditional Constant* algorithms by Wegman and Zadeck [24]. The constant propagation is path sensitive: it takes simple comparisons on booleans and strings into account. The flag variables are initialized in the `main` method as done in the filter class (for example, `stripIt` from `BookmarksRemover` is initialized to `false`).

    This results in specialized versions of the methods depending on their invocation context, and it eliminates infeasible control flow at invocation sites.

For the schema `cards.xsd` and the `NamesFilter` class from Example 2, we extract the element names {`cards`, `card`, `name`, `email`, `phone`} and the flag variables {`is_name`, `has_phone`}. (Note that extracting imprecise collections here may affect performance and precision of the analysis but not soundness.) This means that 20 copies of the `startElement` method are considered (although 15 of them are subsequently eliminated as unreachable code):

```
public void startElement[localName="card",
                         is_name=false,
                         has_phone=false](...) {
  is_name = localName.equals("name");
  if (is_name)
    name = new StringBuffer();
  if (localName.equals("phone"))
    has_phone = true;
}

public void startElement[localName="name",
                         is_name=false,
                         has_phone=false](...) {
  is_name = localName.equals("name");
  if (is_name)
    name = new StringBuffer();
  if (localName.equals("phone"))
    has_phone = true;
}

public void startElement[localName="phone",
                         is_name=false,
                         has_phone=false](...) {
  is_name = localName.equals("name");
  if (is_name)
    name = new StringBuffer();
  if (localName.equals("phone"))
```

[2] A lazy evaluation strategy can be used to avoid explicitly producing all possible combinations.

```
    has_phone = true;
}

.
.
.
```

(The methods are here written with augmented method names and without the SAXExceptions.) After the constant propagation and unreachable code elimination, the methods have been specialized:

```
public void startElement[localName="card",
                         is_name=false,
                         has_phone=false](...) {
  is_name = false;
}

public void startElement[localName="name",
                         is_name=false,
                         has_phone=false](...) {
  is_name = true;
  name = new StringBuffer();
}

public void startElement[localName="phone",
                         is_name=false,
                         has_phone=false](...) {
  is_name = false;
  has_phone = true;
}

.
.
.
```

When we now run the well-formedness and validity analysis as in Section 3, precision is sufficient for the `NamesFilter` and `BookmarksRemover` examples. In fact, for `NamesFilter`, specializing with respect to only the element names is sufficient, whereas correctness of `BookmarksRemover` also depends on the flag variables. Applying the method specialization on `BookmarksRemover` using a dummy input schema results in the following methods, among others:

```
public void startElement[qName="channel",
                         stripIt=false](...) {
  if (?)
    stripIt = true;
  if (!stripIt)
    super.startElement(..., "channel", ...);
}

public void startElement[qName="channel",
                         stripIt=true](...) {
  if (?)
    stripIt = true;
  if (!stripIt)
    super.startElement(..., "channel", ...);
}

public void endElement[qName="channel",
                       stripIt=false](...) {
  super.endElement(..., "channel");
}

public void endElement[qName="channel",
                       stripIt=true](...) {
  stripIt = false;
}

.
.
.
```

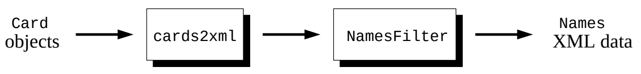
Running the analysis from Section 3 on this code reveals that it may produce output that is not well-formed (that is, start and end tags are not balanced); indeed, the program only runs correctly under the assumption that `channel` elements are not nested.

The success of this method specialization approach depends on several assumptions discussed in earlier sections. In particular, SAX programs that involve many flag variables may lead to a blow-up in program size, and reasoning about well-formedness and validity may be more intricate, requiring more sophisticated techniques than focusing only on element names and flag variables. Still, according to the small study mentioned before [11], SAX programs tend to follow the patterns covered by the approach suggested here.

4.3 Filter Pipelines

Let us now return to the issue of pipelines of filters. Pipelines can be handled rather elegantly through the use of XML graphs for modeling both input and output of individual filters. Assume that we have a pipeline consisting of n filters, $F_1, \ldots, F_n$, a schema S_0 describing the initial input, and a schema S_n describing the final output. We now run the analysis of F_1 using S_0 as input schema, which gives us an XML graph X_1 describing the possible output of the first filter. Rather than requiring a schema for describing the possible input of F_2 we may now simply bypass that step and use X_1 for the purpose (recall from Section 4.1 that the input schema is converted via an XML graph anyway). This process is repeated until the last filter, F_n, whose output XML graph X_n is compared against the schema S_n. Thus, the analysis is inherently compositional, assuming that the filter pipeline is known statically.

For instance, this would allow us to check validity of the output of pipelining Example 1 and Example 2:



As an alternative (or supplementary) strategy we could annotate each intermediate pipeline stage with a schema, much like the use of optional schema annotations in XACT [12].

4.4 Remaining Issues

The previous sections have focused on the most central aspect of XML well-formedness and validity: the structuring of element tags. Extending the analysis to also model character data events and attributes is less elegant due to the peculiarities of SAX.

Character data is represented by *text* nodes in the XML graphs. In the conversion to event producer code, a *text* node labeled with a regular language describing the possible values can be converted into an invocation of the `characters` method. However, since contiguous character data in SAX may be reported as multiple events as mentioned earlier, one *text* node must be converted into a loop containing an invocation of `characters` labeled with *all possible substrings* of the character data in order to preserve soundness. Clearly, this will incur a loss of precision in some cases. A possible way around that could be to identify occurrences of the pattern used in Example 2 for collecting the substrings in a `StringBuffer` and for these cases directly model these `StringBuffers` as the regular languages from the *text* nodes. Again, experiments will show whether this suffices in practice.

To model attributes, we need a way of converting *attribute* nodes in XML graphs into additional information on invocations of `startElement`. This is naturally connected to the discussion of the `Attributes` interface in Section 3.4, and we leave this aspect of the analysis to future work.

5. Conclusion

We have exposed the challenges that must be tackled in order to provide static analysis of event-based XML processing applications that use general purpose programming languages. Concretely, this paper has focused on SAX. The challenges include reasoning about sequences of SAX events both as input and as output, flow sensitive string computations, attribute maps, and field variables in Java.

In addition to discussing the challenges, we have outlined a strategy for a particular program analysis that may serve as a starting point. To summarize, the key ideas suggested here are the following, which build on top of the existing program analysis technique for Java Servlets and XACT:

- producers of SAX events can be modeled via a translation into string-based output stream operations (Section 3); and
- SAX filters can be modeled by a reduction to event producers via a translation from XML schemas to program code that simulates the possible events, together with cloning-based path sensitive method specialization (Section 4).

Besides fitting naturally with the existing analysis techniques, the use of XML graphs for representing sets of XML documents makes the analysis inherently compositional, making it capable of also handling filter pipelines.

The next step is to implement the central parts of the analysis and evaluate the performance and precision on real SAX applications to be able to prioritize the efforts on the remaining challenges. Some progress in this direction is reported in the master's thesis by Jacobsen [11]. This includes an implementation of a simplified version of the conversions from SAX event producers to servlets and from schemas to flow graphs, and additionally a preliminary study of SAX programs found on the web, which has confirmed the assumptions made in this paper about how SAX is typically being used. Larger case studies and a complete implementation, however, remain as future work. Another interesting direction is to consider well-formedness and validity analyses for other event-based XML frameworks, in particular STX [5].

References

[1] David Brownell. *SAX2*. O'Reilly & Associates, January 2002.

[2] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '90*, June 1990.

[3] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In *Proc. 4th Asian Symposium on Programming Languages and Systems, APLAS '06*, November 2006.

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[5] Petr Cimprich et al. Streaming transformations for XML (STX). Working Draft, April 2007.

[6] Daniela Florescu, Chris Hillery, Donald Kossmann, Paul Lucas, Fabio Riccardi, Till Westmann, Michael J. Carey, and Arvind Sundararajan. The BEA streaming XQuery processor. *VLDB Journal*, 13(3):294–315, 2004.

[7] Alain Frisch and Keisuke Nakano. Streaming XML transformation using term rewriting. Presented at Programming Language Technologies for XML, PLAN-X '07.

[8] Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *Proc. ACM SIGMOD International Conference on Management of Data, SIGMOD '03*, June 2003.

[9] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[10] Bárður Háskor. Analysis of string expressions. Master's thesis, Department of Computer Science, University of Aarhus, 2007.

[11] Anders Jacobsen. Analyse af SAX applikationer. Master's thesis, Department of Computer Science, University of Aarhus, 2007. (In Danish).

[12] Christian Kirkegaard and Anders Møller. Type checking with XML Schema in XACT. Technical Report RS-05-31, BRICS, 2005. Presented at Programming Language Technologies for XML, PLAN-X '06.

[13] Christian Kirkegaard and Anders Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*. Springer-Verlag, August 2006. Full version available as BRICS RS-06-10.

[14] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[15] Donald E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11:269–289, 1967.

[16] Koichi Kodama, Kohei Suenaga, and Naoki Kobayashi. Translation of tree-processing programs into stream-processing programs based on ordered linear type. In *Proc. 2nd Asian Symposium on Programming Languages and Systems, APLAS '04*, November 2004.

[17] Yasuhiko Minamide and Akihiko Tozawa. XML validation for context-free grammars. In *Proc. 4th Asian Symposium on Programming Languages and Systems, APLAS '06*, November 2006.

[18] Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. *ACM Transactions on Programming Languages and Systems*, 29(4), July 2007.

[19] Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. 10th International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[20] Anders Møller and Michael I. Schwartzbach. XML graphs in program analysis. In *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '07*, January 2007.

[21] Keisuke Nakano and Shin-Cheng Mu. A pushdown machine for recursive XML processing. In *Proc. 4th Asian Symposium on Programming Languages and Systems, APLAS '06*, volume 4279 of *LNCS*. Springer-Verlag, November 2006.

[22] Stefanie Scherzinger and Alfons Kemper. Syntax-directed transformations of XML streams. Presented at Programming Language Technologies for XML, PLAN-X '05.

[23] Micha Sharir and Amir Pnueli. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[24] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 12(2):181–210, 1991.

[25] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '04*, June 2004.