BRICS RS-06-16 A. Møller: Static Analysis for Event-Based XML Processing

BRICS

Basic Research in Computer Science

Static Analysis for Event-Based XML Processing

Anders Møller

BRICS Report Series RS-06-16

ISSN 0909-0878 October 2006

BRICS
Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK–8200 Aarhus N
Denmark
Telephone: +45 8942 9300
Telefax: +45 8942 5601
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/06/16/`

Static Analysis for Event-Based XML Processing
(Work-In-Progress Paper)

Anders Møller

BRICS, Department of Computer Science,
University of Aarhus, Denmark
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 and presents some considerations for creating program analyses for SAX applications. In particular, we consider the problem of statically guaranteeing that a given SAX application always produces only well-formed and valid XML output.

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 [13].) 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 or adding, for example, schema-based type annotations.

The approach suggested here builds upon existing work on static analysis for the XACT system [9, 7], analysis for Java Servlets [8], and analysis of string operations in Java [4]. It would of course be an interesting challenge to instead try building on alternative approaches, for example regular expression types [6], but we leave that to others.

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 examples. Section 3 outlines a well-formedness and validity analysis of Java programs that produce SAX events as output, and Section 4 considers the – apparently 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.

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 [12]) provide inspiration for developing type checking or static analysis for the domain-specific event-based language STX [5].

2 The SAX 2.0 Framework and its 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 special kinds of 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>
```

For instance, the output could be the following sequence of events for a single `Card` object:

start document
start prefix mapping "" ↦ `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.

Naturally, filters can 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;
    }
  }
}
```

This 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 needs to be 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 in fact 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 (as the two booleans 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.

(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 variables and simple string comparisons.

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 [8] and described briefly below. (See also the recent work by Minamide and Tozawa [11].) 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 should be evident:

```
void cards2xml(Collection<Card> cards, PrintWriter out) {
  out.print("<cards>");
  for (Card c : cards) {
    out.print("<card id=\""+escapeXML(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 [8] 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.

The following sections describes the approach in more detail. The structure of the analysis is illustrated in Figure 1.

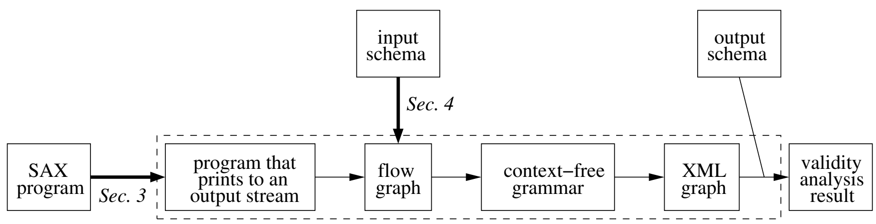


Fig. 1. Structure of the analysis. The dashed box contains parts that are described in earlier papers [8, 7].

3.1 Analysis of String Operations

The string analysis, 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 analysis has some well-known limitations regarding precision, but related work [3] shows that it can smoothly be extended with standard techniques for better heap modeling [2] and context sensitivity [14].

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 `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. In a forthcoming version of the string analyzer, this can be controlled by the programmer using a string type annotation feature.)

3.2 Analysis of String-based Output Streams

The overall structure of the analysis of output streams, as presented in [8], 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.

This 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,

[1] In [8], return nodes are defined as a special kind of nop nodes.

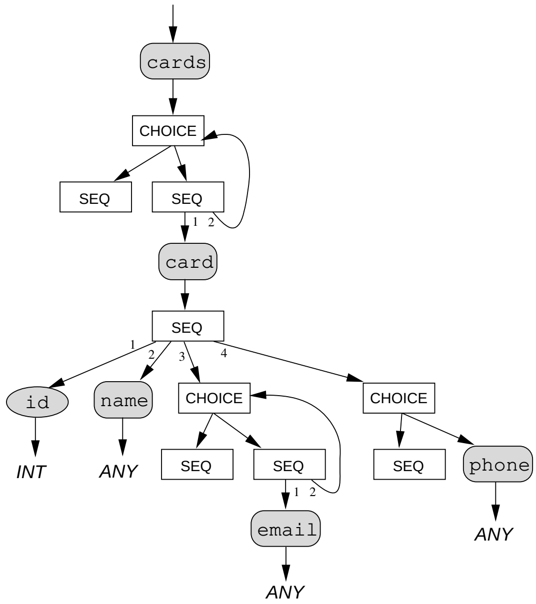


Fig. 2. XML graph for Example 1.

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 [10]. 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 (also called a *summary graph* in earlier papers). The XACT project [7] 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 2 using the graphical notation explained in [8]. 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 [8] for a more formal description.) The *text* node *INT* is here the regular language for integers described in Section 3.1, and *ANY* is the set of all Unicode strings.

3.3 Remaining Issues

One remaining problem is the modeling of the namespace mapping events. A simple approach for also handling those events would be to treat `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`.

Element name strings can be computed dynamically and used for producing events in the following style:

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

Existing string analyses may be able to find out that the only possible values of `foo` 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 `foo` always has the same value when `startElement` and `endElement` are invoked. One approach to achieve this could be 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, but it may also be interesting to look for a more general solution. Experiments with real applications will show whether this is altogether a problem occurring in practice.

The challenges (3) and (4) remain. Regarding the character array intervals, the following observations have been made on a tiny study of SAX programs: First, the typical case where a string is converted into a character array interval (as in Example 1) is easily recognized and modeled. Second, most other character array intervals presumably 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.

Regarding construction of objects of type `Attributes`, it appears reasonable to concentrate the effort on the standard implementing class `AttributesImpl` as other implementations are uncommon. If furthermore only considering the method `addAttribute` (of course, some experiments should be made to find out to what extent other methods are being used), then these objects can simply be represented as finite maps from attribute names to attribute values where both are modeled as regular languages. (Since the string analysis already operates with regular languages, this is the obvious choice here.) A lesson learned from developing analyses for XSLT [12] that could also be useful here is that a good balance between analysis precision, performance, and simplicity can be obtained from a preliminary study of a large collection of real programs.

4 Analyzing SAX Event Consumers

To reason about applications that consume SAX events, such as content handlers or filters like Example 2, 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 a flow graph that corresponds to the possible events being generated by a SAX parser reading valid input. This flow graph then acts as a “main” method that invokes the appropriate event handlers. (As mentioned in Section 3.2, a flow graph is divided into fragments, each abstractly describing a method.) The combined program – consisting of this main method and the actual code to be analyzed – is then processed as described in the previous sections.

Let us now explain in more detail how this will work and identify the remaining problems. First, the XACT project provides a translation from schemas written in XML Schema into XML graphs [7]. For instance, the XML graph being generated for the schema `cards.xsd` from Section 2 is essentially the one shown in Section 3.2. Each node in the XML graph is then translated into a flow graph fragment, which abstractly represents a method, as follows:

– An *element* node results in a method that first calls `startElement` (using an invoke node), then it calls the method that corresponds to its content node, and finally it calls `endElement`. To properly model the element names, which are described by regular languages in XML graphs, we augment the generated invoke nodes with such regular languages.
– 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, becomes a method that contains a single invoke node with edges to the corresponding methods.

(We defer the modeling of *attribute* and *text* nodes to Section 4.1.) 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 resulting flow graph effectively combines the input schema with the program code.

As an example, the schema `cards.xsd` is converted into flow graph fragments that can be described by the following (abbreviated) pseudo-code:

```
main {
 invoke[startDocument]
 invoke[element1]
 invoke[endDocument]
 return
}

element1 {
 invoke[startElement("cards")]
 invoke[choice1]
 invoke[endElement("cards")]
```

```
  return
}

choice1 {
  invoke[seq1,seq2]
  return
}

seq1 {
  return
}

seq2 {
  invoke[element2]
  invoke[choice1]
  return
}

⋮
```

Together with the flow graph fragments produced from the code from Example 2 we can now continue the analysis as in Section 3.

4.1 Remaining Issues

The translation from schemas to flow graphs explained above is a key to addressing challenge (5), however, there still are a few problems.

First, as explained above, some invoke nodes (those with target `startElement` or `endElement`) now use regular languages to describe the possible string values that may flow as method arguments. What we essentially need to maintain this information throughout the rest of the analysis is to extend it with a suitable degree of *context sensitivity* [14]. A starting point could be to analyze the `startElement`/`endElement` handlers polyvariantly with a flow graph fragment copy for each of the finitely many regular languages that arise. Practical experiments are necessary to find out whether this is enough (at least it would be enough to handle Example 2), but fortunately the literature provides plenty of general techniques for making context sensitive analyses.

To model attributes, we need a way of converting *attribute* nodes in XML graphs into additional information on the invoke nodes that have target `startElement`. This is naturally connected to the discussion of the `Attributes` interface in Section 3.3, and we leave this aspect of the analysis to future work.

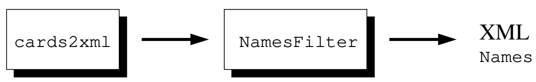
Character data, which is represented by *text* nodes in the XML graphs, similarly needs to be handled in the conversion to flow graphs. This is also connected to the discussion of `characters` events in Section 3.3. A *text* node is labeled with a regular language describing the possible values. Again, we augment the relevant flow graph nodes (invoke nodes with target `characters`) with regular languages describing the possible character data values, and analyze the `characters` event handlers polyvariantly. However, since contiguous character data may be reported as multiple events as mentioned earlier, one *text* node must be converted to a loop containing an `invoke` node augmented with a regular language describing *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 `StringBuffer`s as the regular languages from the *text* nodes. Again, experiments will show whether this suffices in practice.

Regarding modeling of fields (challenge (6) in Section 2), we can take advantage of the following pattern that appears to be common in SAX filters: the fields are typically modified only from inside the filter (perhaps they are declared `private`), and by only one thread. This means that each field can be modeled flow sensitivity as a global variable. For programs not adhering to this pattern, a significant loss of analysis precision at present seems inevitable.

A part of the problem with conditionals (challenge (7)) has already been solved by making the analysis context sensitive with respect to element names. However, it is easy to construct plausible SAX programs where path sensitivity is crucial. We leave also this aspect to future work.

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 involve a schema describing the possible input of F_2 we may now simply bypass that step and use X_1 for the purpose. 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 [7].

5 Conclusion

We have exposed some of the considerable challenges that must be tackled in order to provide static analysis of event-based XML processing applications. 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 a general-purpose object-oriented language.

In addition to discussing the challenges, we have outlined a preliminary 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; and
– consumers of SAX event, in particular filters, can be modeled via a translation from XML schemas to flow graphs.

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.

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, July 2004.
6. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
7. 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.
8. 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.
9. 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.
10. Donald E. Knuth. A characterization of parenthesis languages. *Information and Control*, 11:269–289, 1967.
11. 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.

12. Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, 2005. Draft, accepted for TOPLAS.
13. 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.
14. Micha Sharir and Amir Pnueli. Two approaches to interprocedural dataflow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

Recent BRICS Report Series Publications

RS-06-16 **Anders Møller.** ***Static Analysis for Event-Based XML Processing*. October 2006. 16 pp.**

RS-06-15 **Dariusz Biernacki, Olivier Danvy, and Kevin Millikin.** ***A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations*. October 2006. ii+28 pp. Revised version of BRICS RS-05-16.**

RS-06-14 **Giorgio Delzanno, Javier Esparza, and Jiří Srba.** ***Monotonic Set-Extended Prefix Rewriting and Verification of Recursive Ping-Pong Protocols*. July 2006. 31 pp. To appear in ATVA '06.**

RS-06-13 **Jiří Srba.** ***Visibly Pushdown Automata: From Language Equivalence to Simulation and Bisimulation*. July 2006. 21 pp. To appear in CSL '06.**

RS-06-12 **Kristian Støvring.** ***Higher-Order Beta Matching with Solutions in Long Beta-Eta Normal Form*. June 2006. 13 pp. To appear in *Nordic Journal of Computing*, 2006.**

RS-06-11 **Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski.** ***An Interface Theory for Input/Output Automata*. June 2006. 40 pp. Appears in Misra, Nipkow and Sekerinski, editors, *Formal Methods: 14th International Symposium*, FM '06 Proceedings, LNCS 4085, 2006, pages 82–97.**

RS-06-10 **Christian Kirkegaard and Anders Møller.** ***Static Analysis for Java Servlets and JSP*. June 2006. 23 pp. Full version of paper presented at SAS '06.**

RS-06-9 **Claus Brabrand, Robert Giegerich, and Anders Møller.** ***Analyzing Ambiguity of Context-Free Grammars*. April 2006. 19 pp.**

RS-06-8 **Christian Kirkegaard and Anders Møller.** ***Static Analysis for Java Servlets and JSP*. April 2006. 22 pp.**

RS-06-7 **Petr Jančar and Jiří Srba.** ***Undecidability Results for Bisimilarity on Prefix Rewrite Systems*. April 2006. 20 pp. Presented at *FoSSaCS 2006*, LNCS 3921:277–291.**

RS-06-6 **Luca Aceto, Willem Jan Fokkink, Anna Ingólfsdóttir, and Bas Luttik.** ***A Finite Equational Base for CCS with Left Merge and Communication Merge*. March 2006. 22 pp.**