Viola: A Verifier For Interoperating Components

Mark Grechanik

The University of Texas at Austin,
Austin TX 78729, USA,
gmark@cs.utexas.edu

Abstract. Two or more components (e.g., objects, modules, or programs) interoperate when they exchange data, such as XML data. Currently, there is no approach that can detect a situation at compile time when one component modifies XML data so that it becomes incompatible for use by other components, delaying discovery of errors to runtime. Our solution, a ***V**erifier for **I**nteroperating c**O**mponents for finding **L**ogic f**A**ults (Viola)* builds abstract programs from the source code of components that exchange XML data. Viola symbolically executes these abstract programs thereby obtaining approximate specifications of the data that would be output by these components. The computed and expected specifications are analyzed to find errors in XML data exchanges between components. We describe our approach, implementation, and give our error checking algorithm. We used Viola on open source and commercial systems and discovered errors that were not detected during their design and testing.

1 Introduction

Components are modular units (e.g., objects, modules, or programs) that interact by exchanging data. Components are hosted on a platform, which is a collection of software packages. These packages export *Application Programming Interfaces (APIs)* through which components invoke platform services to access and manipulate data. For example, an *eXtensible Markup Language (XML)* [3] parser is a platform for XML data; it exports APIs that different components can use to access and manipulate XML documents.

Two or more components interoperate when they exchange information [9]. It is conservatively estimated that the cost of programming errors in component interoperability just in the capital facilities industry in the U.S. alone is $15.8 billion per year. A primary driver for this high cost is fixing flaws in incorrect data exchanges between interoperating components [10].

Type checking algorithms can be used to verify the correctness of operations on types of exchanged data within a single program statically. However, there are many situations where the static type checking of interoperating components is not attempted, resulting in the run-time discovery of type errors. In a model shown in Figure 1, `J` and `C` are components (say a Java and C++ components respectively) that interact using XML data $\mathtt{D}_1$ and $\mathtt{D}_2$. Component `J` reads in data $\mathtt{D}_1$, modifies it, and passes it as data $\mathtt{D}_2$ to the component `C`. Component `C` reads in the data $\mathtt{D}_2$ expecting it to conform to some specification S (i.e. a schema [1]). Since J outputs the data D_2 before C accesses it, concurrency is not relevant. However, because of design or programming errors, the component J outputs the data D_2 as conforming to a different specification S′, which is not explicitly stated in any design documents. Since S′ is different from S, a runtime error may be issued when C reads D_2.

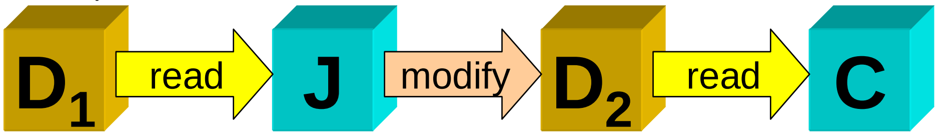


Fig. 1. A model of component interoperability.

This problem is typically addressed by using an XML parser to validate that the data D_2 conforms to the specification S when J produces this data. If the data does not conform to this specification, then J throws a runtime exception. Obviously, it is better to predict possible errors at compile time rather than to deal with them at runtime.

In reality, the situation is even more complicated. Using specifications for validating XML data is not often attempted because it degrades components performance, and it even leads to throwing exceptions when there may not be any runtime errors [33][32]. Suppose that the component J deletes all instances of some data element thus violating the specification S that requires this element be present in the data D_2. If either of these components validates this incorrect data D_2 against the specification S, then it will issue a runtime error. However, when executed, the component C may never attempt to access the deleted data element, and therefore, no exception will be thrown if the validation step is bypassed. It is important to know what paths (data elements) components J and C access in the specifications, and if these paths do not intersect, then components J and C may still interact safely even if the data D_2 does not conform to its specification.

Although it is known in advance that components exchange data, it is not clear how to detect operations at compile time that lead to possible runtime errors. Given interoperating components J and C producing and exchanging data D_2 at runtime and the specification of this data S, the main safety property is defined as ensuring D_2 conforms to S. A secondary safety property is defined as navigation paths to data elements in S should not intersect with paths to elements added and deleted by interoperating components provided that specifications are not used at runtime to validate XML data. Our goal is to verify whether interoperating components satisfy these safety properties at compile time. Currently, no compiler checks interoperating components for violations of these properties, even when components are located within the same program.

Our solution is a ***V**erifier for **I**nteroperating c**O**mponents for finding **L**ogic f**A**ults (Viola)* that deals with components exchanging XML data. Viola uses the well-known *abstract-verify-refine* verification paradigm [13][17][20][15]. During the abstraction step Viola creates models of the source code of components and approximate specification of the data that these components exchange. To do that, Viola takes the com-

[1] A schema is a set of artifact definitions in a type system that defines the hierarchy of elements, operations, and allowable content.

ponents source code, specifications for the XML data used by these components, and *Finite State Automata (FSAs)* that model abstract operations on data with low-level platform APIs. Abstract operations include navigating to data elements, reading and writing them, adding and deleting data elements, and loading and saving XML data. These FSAs are created by expert programmers who understand how to use platform APIs to access and manipulate data.

Viola uses control and data flow analyses along with the provided FSAs to extract abstract operations from the component source code for each path in the component's *Control Flow Graph (CFG)*. Next, these operations are symbolically executed to compute approximate specifications of the data that would be output by different execution paths of the components. During the verification step, the computed and expected specifications are compared to determine if they match each other. If a mismatch between them is found, then it means that one component modifies the data incorrectly so that runtime exceptions may be thrown by other components using this incorrect data. To confirm that, Viola executes the refinement step during which it analyzes paths to data elements accessed and modified by these components to determine whether the schema mismatch results in actual errors. Sequences of operations leading to errors are reported to programmers.

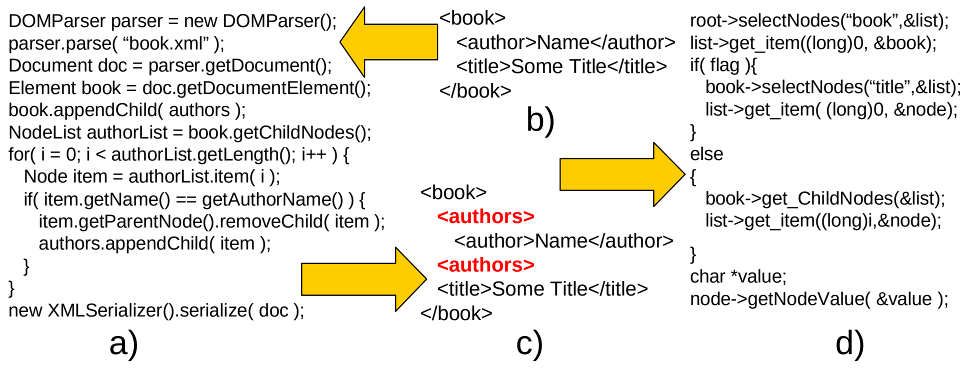
Viola's conservative static analysis mechanism reports potential errors or ensures their absence for a system of interoperating components. We tested Viola on open source and commercial systems, and we detected a number of known and unknown errors in these applications with good precision thus proving the effectiveness of our approach.

2 A Motivating Example

Shown in Figure 2 are fragments of Java (Figure 2a) and C++ code (Figure 2d) for two respective components that interoperate using XML data (Figure 2b-c). This is a running example that we use to illustrate our approach throughout this paper. Block arrows show the flow of XML data between components. Variations of these code fragments are used in many open source and commercial applications. The Java component uses `Xerces` DOM parser API to read in and modify XML data shown in Figure 2b. This XML data describes the attributes of a book that include the author and the title. The Java component modifies the structure of the XML component by adding the tag `authors` as a child element of the root `book` and moving the `author` element under the tag `authors`. The resulting XML data is shown in Figure 2c.

The C++ component shown in Figure 2d reads in the XML data and depending on the value of the boolean variable `flag`, either returns the title or the name of the author of a book. The writer of this component assumes that a book has a single author, and the structure of XML data corresponds to the one shown in Figure 2b. When the Java component modifies this data and the value of the boolean variable flag is false, the C++ component code throws a run-time exception because the element `author` is not present in the XML data under the root element `book`.

In this example, schemas are not used to validate the XML data at runtime. If they were used, then exceptions would be thrown during runtime validation of XML data



```
DOMParser parser = new DOMParser();
parser.parse( "book.xml" );
Document doc = parser.getDocument();
Element book = doc.getDocumentElement();
book.appendChild( authors );
NodeList authorList = book.getChildNodes();
for( i = 0; i < authorList.getLength(); i++ ) {
   Node item = authorList.item( i );
   if( item.getName() == getAuthorName() ) {
      item.getParentNode().removeChild( item );
      authors.appendChild( item );
   }
}
new XMLSerializer().serialize( doc );
```

```
<book>
   <author>Name</author>
   <title>Some Title</title>
</book>
```

```
<book>
   <authors>
      <author>Name</author>
   <authors>
   <title>Some Title</title>
</book>
```

```
root->selectNodes("book",&list);
list->get_item((long)0, &book);
if( flag ){
   book->selectNodes("title",&list);
   list->get_item( (long)0, &node);
}
else
{
   book->get_ChildNodes(&list);
   list->get_item((long)i,&node);
}
char *value;
node->getNodeValue( &value );
```

Fig. 2. Java (a) and C++ (d) components that interoperate using XML data (b) and (c).

either in the the Java component after it modified the data, or in the C++ component before it reads the data. If XML data is not validated at runtime, then exceptions will be thrown when certain APIs are called to access data elements. Either way, runtime errors occur whether XML data is validated or not.

Even with this simple example it takes a considerable amount of time to find errors. Several factors are involved: knowing the structure of the input data and how changes made by components affect it, using platform APIs correctly and translating API calls at compile time into changes that would be made to data, and knowing the order in which components execute. The temporal dependency between the order of component execution and the visibility of errors makes catching errors especially difficult. If the C++ component executes before the Java component, then it would operate on the correct XML data shown in Figure 2b. However, if the Java component executes before the C++ component, then it would modify the data into an instance shown in Figure 2c, and make it incompatible for the C++ component. These factors add to the complexity of interoperating components, and make it difficult to catch errors at compile time.

3 The Architecture of Viola

Viola's architecture and process are shown in Figure 3. The steps of the Viola process are presented with numbers in circles. The names of components and data are taken from the model shown in Figure 1. The input to the architecture is the `J`'s and `C`'s components source code `(1)`. The `EDG C++` and `Java front ends` [7] parse the source code of the components and output *Abstract Syntax Trees (ASTs)* `(2)`. The *Analysis Routines (ARs)* perform control and data flow analyses on the ASTs in order to determine sequences of API calls that can be replaced with abstract operations. ARs also input FSAs for abstract operations `(3)`, and check to see if sequences of API calls retrieved from the source code are accepted by these FSAs. If a sequence of API calls is not recognized, or some abnormalities in using these APIs are detected, then errors in using APIs are reported to programmers `(4)`.

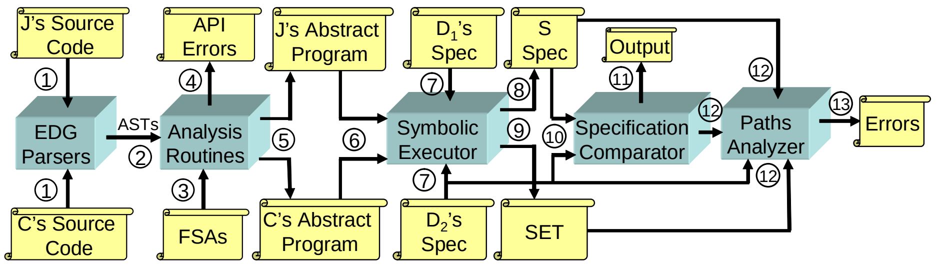


Fig. 3. Viola's architecture and process.

Running ARs results in abstract programs for C and J components (5). Abstract programs represent sequences of abstract operations on the XML data D_1 and D_2. The *Symbolic Executor* takes these abstract programs (6) and the specifications of the XML data D_1 and D_2 as its input (7) and outputs the specification S (8) resulting from executing J's abstract program symbolically on the D_1's specification, and *Symbolic Execution Trees (SETs)* (9) that represent symbolic executions of these abstract programs on the data specifications. SETs are graphs characterizing the execution paths followed during the symbolic executions of a program. Nodes in these graphs correspond to executed statements, and edges correspond to transitions between statements. Viola process steps 1-9 constitute the abstraction step of our verification approach.

The *Specification Comparator (SC)* compares the computed specification S with the D_2's specification (10). If these specifications are equivalent, then the SC reports that components passed the verification (11). Viola process steps (10) and (11) constitute the verification step of our approach.

If the verification step fails, we refine our analysis to obtain diagnostic messages of higher precision. To do that, the *Paths Analyzer (PA)* analyzes the SETs, D_2's specification and the specification S (12) to determine various errors in the paths computed by components to accessed and modified data elements (e.g., components access, delete, or add elements that do not exist in the specifications for the data). PA reports the discovered errors to programmers (13).

A component may or may not validate an XML document at runtime using its specification. If parsing is performed with verification, then a run-time error is thrown at parse time; otherwise, a run-time error may be thrown when XML data is processed within a component. Since our goal is to predict runtime errors or ensure their absence at compile time, we consider parsing XML data without using data specifications.

4 Errors

To summarize errors that Viola catches in interoperating components, we classify errors into the following general categories:

– *Path-Path (PP)* errors occur when a component accesses elements that may be deleted by some other components.

  PP errors occur in components that access data elements that are deleted by some other components (PP-1) and by components that read or write wrong elements (PP-2). PP-1 errors are execution-order-dependent and therefore are difficult to find using testing or manual code inspection. If some component deletes data elements after some other component accesses these elements, then the execution proceeds correctly. However, if the order of the execution is reversed, then an exception will be thrown by a component that accesses a previously deleted element. PP-2 errors occur when one component navigates to a wrong data element and reads its value by using sequence numbers of elements for navigating rather than their names. If some other component inserts a data element into the navigation path of the first component, then the result of interference of these operations is that the first component accesses and reads the value of a different data element from what was intended.

– *Path-Schema (PS)* errors occur when components attempt to access, delete, or add elements that do not exist in the schemas for the data (PS-2), or when components violate bounds set by schemas on data elements as a result of executing operations on data (PS-1).
  PS-1 errors occurs when components violate constraint bounds set by a schema. Suppose that a schema defines the value of the `minOccurs` attribute for a data element to be equal to one, however, a component may delete all instances of this element. Some other component may execute code that was written based on the assumption that at least one instance of this data element should be present in the XML data. This situation may also lead to execution-order-dependent runtime errors.
– *API errors* that result from incorrect uses of APIs.
  Mastering APIs for accessing and manipulating data often requires programmers to spend long periods of time learning dependencies between APIs and objects that are created as results of their calls [31][34]. One of common mistakes is that programmers use incorrect APIs in the sequences of calls designed to perform operations on data. Given that the knowledge of how to use APIs correctly is encapsulated in the FSAs that expert programmers build for abstract operations, Viola can flag sequences of API calls that cannot be accepted by FSAs as erroneous at compile time. It may also be that the flagged sequence of API calls is correct, and no FSA was provided to Viola to validate this sequence. In this case experts will add an FSA to the Viola FSA database, and these sequences will be accepted from that moment on.
  Sometimes programmers forget to make components save their changes to data (e.g., a `return` statement may be executed before the `Save` operation in some execution path). Technically, it is not an incorrect use of API, but rather omission of a crucial operation that makes changes to data persistent. The data remains consistent after operations are executed; however, changes made by the component that does not save the data will be lost. Viola reports these situations to programmers at compile time helping them to find and debug potential logical faults.

Below are examples of error warnings that Viola issues to programmers after it analyzes interoperating components:

PP-1: At line `23` component `C` accesses element ⟨`book`, `author`⟩ that may be deleted by the component `J` at line `122`.

PP-2: At line `23` component `C` may read a wrong element located under path ⟨`book`⟩ because component `J` modifies elements under this path at line `122`.

PS-1: At line `23` component `C` may delete all instances of the element ⟨`book`, `author`⟩, however, at least one instance of this element is required by the schema `S`.

PS-2: At line `23` component `C` accesses element ⟨`book`, `royalties`⟩, however, this element is not defined by the schema `S`.

5 The Models

Viola builds a model of a program by abstracting its operations on data, and then symbolically executes these operations on the specification of the data. We describe program abstractions and specify the formal framework for building these abstractions.

5.1 Program Models

A basic programming model for interoperating components is shown in Figure 1. Our goal is to verify at compile time the main safety property defined as the data $\mathtt{D}_2$ conforms to the specification `S`, and a secondary safety property defined as navigation paths to data elements in `S` should not intersect with paths to elements added and deleted by interoperating components. The problem is to determine at compile time how operations that access and modify data violate this property. These operations are navigating to data elements, reading and writing data, adding and deleting data elements, and loading and saving data, designated as `Navigate`, `Read`, `Write`, `Add`, `Delete`, `Load`, and `Save` respectively.

These abstract operations are implemented in components using concrete low-level platform APIs. In general, APIs are complex and the client code that uses the APIs is hard to write. It is rare that a one-to-one correspondence exists between abstract operations and APIs. Programmers have to execute a sequence of API calls in order to accomplish each of our abstract operations [31][34]. For example, the `getDocumentElement` API returns a node in the internal representation of XML by *Document Object Model (DOM)* parsers. This internal node is used when calling other APIs as shown in Figure 2a.

In Viola we use two abstraction techniques: the cone of influence reduction and data abstraction [18]. Program abstractions represent source code of interoperating components using abstract operations. We observe that a majority of statements and operations in programs are irrelevant to accessing and modifying XML data. Viola abstracts away specifics of APIs and program variables that do not affect XML data, and transforms programs into sequences of abstract operations. This is known as the cone of influence reduction technique that decreases the number of states of components by focusing on a subset of their variables related to the given specification and eliminating other variables. Data abstractions represent actual data with smaller generalized specifications. We discuss data abstractions in the next section.

```
Load book.xml
        Navigate root
        Add e1
        Loop
                tmp = child
                Delete child
                Navigate e1
                Add tmp
        end loop
Save book.xml
```

a)

```
Load book.xml
        Navigate root
        if
                Navigate “title”
        else
                Navigate e2, seqnumber
        end if

        Read var
Save book.xml
```

b)

Fig. 4. Program abstractions for Java a) and C++ b) components shown in Figure 2a and Figure 2d respectively.

Program abstractions are obtained from source code by analyzing it and mapping sequences of platform APIs to the abstract operations. Examples of program abstractions for the Java and C++ components from Figure 2a and Figure 2d are shown in Figure 4a and Figure 4b respectively. In general, object names (e.g., “`title`”) are not constants; they are expressions whose values are computed at runtime. In program abstractions, these names are replaced with symbolic variables (e.g., `root`, `child`, `element1`, and `e2`) as it is shown in Figure 4. With program abstractions we lose precision, however, the number of program states is significantly reduced.

5.2 Schemas

Data abstractions are achieved with specifications for XML data that are expressed as XML schemas. We consider XML schemas [8] in this paper because XML is the lingua franca of data exchange, and because XML schemas are also used to model non-XML data (e.g., relational databases [30] and PDF files [1]).

XML schemas are recorded in the XML format [8] and each schema has the root specified with the `<schema>` element. Data elements are specified with the `<element>` and with the `<attribute>` tags. Each data element is defined by its name and its type. Elements can be either of simple or complex types. Simply stated, complex element types support nested elements while simple types are attributes and elements of basic types (e.g., `integer`, `string`, or `float`).

Elements may have two kinds of constraints. First, values of elements may be constrained. The second kind of constraints specifies bounds on the number of times that a specific element may occur as a child of some element. These bounds are specified with the `minOccurs` and `maxOccurs` attributes of the ⟨`element`⟩ tag.

Elements can be grouped in a sequence if they are children of the same parent element. Attributes of the same element can also be grouped in a sequence. Each element or attribute in a sequence is assigned a unique positive integer sequence number. This number can be used to access elements or attributes instead of using their names.

We represent schemas using graphs, and we use this formalism for comparing different schemas in order to detect discrepancies leading to runtime errors. Let `T` be finite sets of type names and F of element and attribute names (labels), and distinct symbols $\diamond \in$ F and $\blacksquare \in$ T. Schemas graphs are directed graphs G = (V, E, L) such that

- V$\subseteq$T$\cup\blacksquare$, the nodes are type names or $\blacksquare$ if the type of data is not known;
- L$\subseteq$F$\cup\diamond$, edges are labeled by element or attribute names or $\diamond$ if the name is not known;
- E$\subseteq$L$\times$V$\times$V, edges are cross-products of labels and nodes. If $\langle l, v_k, v_m \rangle \in$ E, we write $v_k \xrightarrow{l} v_m$. Nodes v_m are called children of the node v_k. Function children: $v \rightarrow v$ returns a collection of children nodes of the given node.
- Bounds for labels are specified with subscripts and superscripts. Subscripts are used to specify bounds defined by the minOccurs attribute, and superscripts designate the bounds specified by the maxOccurs attribute.

Examples of graphs for two different schemas are shown in Figure 5. These schemas describe instances of XML data shown in our motivating example in Figure 2b and Figure 2c. Each graph has the special node labeled root $\in$ V, where root represents a collection of the root elements. Function root: G $\rightarrow v$ returns a collection of root elements $v \in$ V of the schema graph G. The XML tag <complexType> specifies that an element is a complex type, and it is not represented in the graph. An empty schema has a single root node and no edges. Each node in a schema graph has a collection of children nodes. If an element has no children, then its corresponding node in a schema graph has an empty collection of children nodes.

A path in a schema graph G = (V, E, L) is defined as a sequence of labels p_G = $\langle l_1, l_2, \ldots, l_n \rangle$, where $v_k \xrightarrow{l_u} v_m$ for $v_k, v_m \in$V and $1 \leq u \leq n$. The symbol $\diamond$ may be used instead of a label in a path if an element is navigated by its sequence number. For example, the symbol $\diamond$ in the path $\langle$book, $\diamond\rangle$ for the schema graph shown in Figure 5a stands for any child element of the element book, and this path may be expanded into two paths: $\langle$book, author$\rangle$ and $\langle$book, title$\rangle$.

Path p_i is a subpath of some other path, p_j, $\mathrm{p}_i \subseteq \mathrm{p}_j$ if and only if all labels of the path p_i are also labels of the path p_j, and the order in which they appear in p_j is the same as they appear in the path p_i. Function length: p $\rightarrow \mathbb{N}$ returns the length of the given path p, that is measured as the number of labels in this path. Function GetLabel: p$\times\mathbb{N}$ returns the label of the path p that is assigned a given sequence number. Function type: $v \rightarrow$s returns the type s$\in$T$\cup\blacksquare$ of the node $v \in$V. Function max: $\mathrm{label}_l^u \rightarrow u$ returns the upper bound u, or ∞ if the upper bound is not specified, and function min: $\mathrm{label}_l^u \rightarrow l$ returns the lower bound l, or zero if the lower bound is not specified.

5.3 The Formal Framework

In order to build program abstractions we need to locate sequences of API calls that correspond to abstract operations. If all sequences of platform API calls for the abstract operations were known in advance, then these sequences can be searched for and replaced with abstract operations. Unfortunately, the multiplicity of data hosting platforms and APIs for accessing and manipulating data makes it difficult to write programs

```
<element name="book">
    <complexType>
        <element name="author"
                 maxOccurs=1/>
        <element name="title"
                 type="string"/>
    </complexType>
</element>
```

```
<element name="book">
    <complexType>
        <element name="authors">
            <complexType>
                <element name="author" minOccurs=1/>
                <element name="title" type="string"/>
            </complexType>
        </element>
    </complexType>
</element>
```

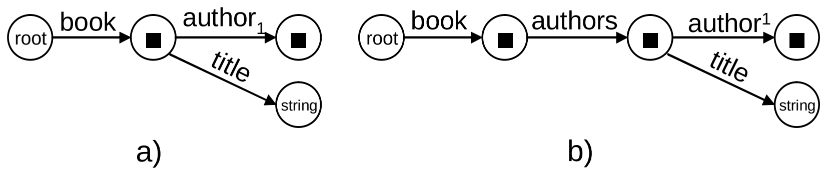


Fig. 5. Examples of graphs for two XML schemas.

that verify interoperating components for every all APIs. We use a formal framework to define these sequences and extract them from source code in a uniform way.

Let Σ be the set of APIs that access and manipulate data, and let Γ be the set of abstract operations, Γ = {`Navigate`, `Read`, `Write`, `Add`, `Delete`, `Load`, `Save`}. Let $\alpha \subseteq \Sigma^*$ be sequences of APIs that access and manipulate data. Partial function $\varphi : \alpha \rightarrow \gamma$ maps a sequence of API calls α to an abstract operation $\gamma \in \Gamma$.

We assume that α, the set of sequences of APIs, is a regular language. Since for each regular language there exists a *Finite State Machine (FSM)* that accepts this language, FSMs are provided for each γ, such that $\varphi^{-1}(\gamma) = \alpha$. If an FSM for an abstract operation γ accepts a sequences of APIs α, then these APIs can be replaced with this abstract operation in the abstract program. By describing sequences of APIs uniformly, Viola can perform its analysis in a platform-independent way.

A trace $\tau \subseteq \Sigma^*$ is a sequence of operations that can be executed by a path in the program. Let τ^* be all possible traces that result from all possible execution paths of the program. If $\tau^* \cap \alpha = \alpha$ and $\varphi(\alpha) = \gamma$, then a subtrace of τ^* can be replaced with the corresponding abstract operation γ. In general, τ^* is not a regular language because it contains traces of function calls. When a function is called, a stack is used to store local variables and the return address. Languages that use stacks are context-free rather than regular. If τ^* is modeled as a context-free language, then there exists a *pushdown automaton (PDA)* that accepts this language. A PDA configuration is described by its current state and all the symbols that the PDA contains in the given state. It was shown that reachable configurations of a PDA form a regular language, and therefore can be represented by some FSM [16][21].

An examples of FSA for the `Navigate` abstract operations for Xerces XML DOM APIs is shown in Figure 6. Circles designate states, and transitions are labeled with

APIs. Double concentric circles stands for final states, and circles with no edges incident on them are the start states. Experts who understand how to use APIs to implement abstract operations construct these FSAs. Once built, these FSAs can be used for building abstract programs from the source code of components.

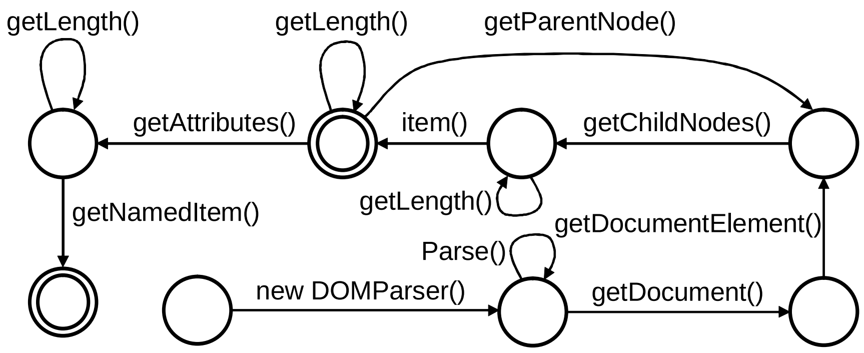


Fig. 6. Example of an FSM for accepting XML DOM APIs for the `Navigate` abstract operation.

In our framework we perform limited analysis of data flow. Specifically, we are only interested in producing separate traces of operations invoked on different objects that represent different data. For example, given two `DOMParser` objects that designate different data, we want to obtain separate traces of methods invoked on these objects. No dataflow analysis is performed on the parameters to these methods. This decision results in better performance of Viola, but the side-effect is more false-positives produced by the compiler. However, we show in our experimental evaluation, the number of false positives is acceptable in practice. In addition, determining that the detected error is false-positive is a simple operation that does not burden programmers a lot.

6 Building Program Abstractions

Here we explain how abstract programs are obtained using components source code and FSAs that model abstract operations. We give a grammar for abstract programs and describe their state variables.

6.1 Extracting Abstract Programs

Program abstractions are obtained from Java and C++ programs using FSAs that map low-level API calls to abstract operations. EDG parser front ends for C++ and Java [7] are used to build *control-flow graphs (CFGs)* from the source code of components. CFGs are graphs whose nodes represent *basic blocks (BBs)* with a statement containing a relevant API call from the set α, and edges between BBs that represent the flow between program statements. Components are partitioned into BBs in the following manner. Statements are read and put into the same BB until either a relevant API call or a branching statement (e.g., `if-then`, `switch-case`, `do-while`) is encountered. Then, this BB is added to the CFG. If the program consists of multiple source files, then the CFGs produced for each of these source files are merged into a single CFG.

Program abstractions are computed for each path in the CFG, where a path is a set of nodes from the CFG connected by edges. For a selected path in the CFG stacks for API calls are created. For each BB containing an API call in the given path, this API is extracted and put on the stack. Every time an API is added, the sequence of API calls on the stack is checked to see if is accepted by any FSA for abstract operations. If such an FSA is found, then this sequence is modeled by the corresponding abstract operation for this FSA, and this operation is put into the abstract program.

Each FSA has a uniquely labeled edge incident on the start node. The label of this edge is an API call that is not encountered anywhere else in the FSAs that implement abstract operations except for the edges incident on the start node. When such an API call is encountered in a CFG, a new stack is created. Several stacks can exist at any given time, and API calls are put on these stacks until they satisfy some FSAs and replaced with the corresponding abstract operations. If no stack can be replaced with an abstract operation, then an API type of error is reported to programmers.

In general, it is an undecidable problem to determine all API calls relevant for a given stack from an arbitrary program. Consider a code fragment where a reference to an object that denotes a data element is put, among other references, into a hash table object, which is passed as a parameter to some function. Inside this function, references to objects are retrieved from the hash table. In a general case it is impossible to determine what exact objects in the calling function the references denote in the hash table inside the called function. To resolve this situation, Viola asks for input from programmers to resolve object references. At this level Viola requires guidance from its users.

6.2 A Grammar of Abstract Programs

A grammar for abstract programs is shown in Figure 7. An abstract program is a sequence of operations and command included within `Load` and `Save` operations. The metavariable `aoper` ranges over abstract operations; `id` ranges over symbolic identifiers (variables); `const` ranges over string and integer constants; `command` ranges over the `state` and `jump` commands as well as assignment expressions, and `body` ranges over program definitions. An abstract program in Viola, `Program`, is a triple `(load, body, save)` of `Load` and `Save` abstract operations and the program body. Assignment expression is also included, allowing us to create global identifiers and assign values to them.

6.3 States and Symbolic Variables

Each program abstraction contains five symbolic state variables: `root`, `current`, `children`, `rw`, and `operation`. The variable `root` contains the names of the root elements of the data, and the variable `operation` contains a current abstract operation performed on data. The variable `operation` is similar to the instruction pointer

Program ::= load $\overline{\texttt{body}}$ save
load ::= Load var
save ::= Save var
body ::= Loop body endLoop | If body endIf | If body else body endIf | $\overline{\texttt{aoper}}$ | $\overline{\texttt{command}}$
aoper ::= Navigate var | Read var | Write var | Add var | Delete var
command ::= state var | jump var | id = var
var ::= id | const
id ::= an identifier
const ::= n | s
n ::= numerical value
s ::= string value

Fig. 7. A grammar for abstract programs

in computers. A set of tuples ⟨current, children⟩ describes elements referenced by a component in a given state and their children elements. In every tuple the variable current contains a path to the element that can be referenced in a given state, and the variable child contains paths to the elements that are children of the element referenced by the variable current. For example, the set of values for the variables are root = {book}, current = {{book}}, and child = {{book, author}, {book, title}} for data shown in Figure 2b in a state when the root element is navigated to. Finally, the rw variable keeps the list of elements and attributes whose values are read or written in a given state.

Symbolic variables are introduced in abstract programs on as-needed basis. Suppose that the names of data elements are not specified as constants in the program shown in Figure 2a, but rather computed at runtime. In this case, these elements are assigned unique symbolic variables. Examples of using symbolic variables instead of the names of data objects are shown in the abstract programs in Figure 4 as elements e1 and e2.

6.4 Description of Abstract Operations

The summary of abstract operations is given in Table 1. Each abstract operation has mandatory and optional arguments. The latter are shown in square brackets. Each data unit has a unique identifier that can be a file name or a sequence of printable characters that uniquely identifies a stream of XML data bytes passed between components.

The operations Load and Save take the identifiers of data as their arguments. These operations mark the beginning and the end of abstract programs. The mandatory parameter of the operation Navigate is the name of a data element or attribute, or its order sequence number in the collection of children elements. The seqnumber optional parameter specifies that the sequence number of the element is used to access it rather than its name. The optional parameter attribute specifies whether an attribute or an element is navigated to, and the optional parameter parent gives the direction of the navigation to parent rather than to children elements. When executed this operation updates the state variables current and child.

OPERATION	DESCRIPTION
`Load identifier`	Start interacting with the XML data referenced by its identifier
`Save identifier`	Update the XML data referenced by its identifier
`Navigate element` [,`seqnumber`] [,`attribute`] [,`parent`]	Navigate to the element whose name is specified by the symbolic variable the optional flag `seqnumber` specifies that the element is located by its sequence number rather than its name, the optional flag `attribute` specifies that the element is an attribute, and the optional flag `parent` specifies that it is navigated to a parent element
`Add element`	Inserts elements specified by the symbolic variable element under the currently navigated elements
`Delete element`	Deletes elements specified by the symbolic variable element that are children elements of the currently navigated element

Table 1. Abstract operations and their descriptions.

The operation `Add` takes names of data elements and adds them under the currently navigated elements updating the `child` state variable. Finally, the `Delete` operation takes names of data elements as its parameter and deletes them from the collection of children of the elements referenced by the variable `current`. When executed this operation updates the `current` and `child` symbolic state variables.

The state of an abstract program is the union of values of the state variables. Abstract operations modify the state of a program by changing values of the state variables. Bookmarking commands "`state <name>`" and "`jump <stateName>`" mark certain states in abstract programs in order to enable execution to return to them from any point when executing these programs. The command "`state <name>`" represents an intermediate object created by platform API calls in the component. For example, operation `doc.getDocumentElement()` creates the transient object book of type `Element` in the Java component shown in Figure 2a. This object represents the root of the data, and it is used in other APIs that navigate down to data elements. We assign some unique name to the state in which this intermediate object is created.

The command "`jump <stateName>`" directs execution of the abstract program to abandon its current state, and this command sets the context associated with some named state and continues to execute the program in the switched state.

Branching statements include boolean predicates specifying the condition under which certain execution paths will be taken. In general, we lack the knowledge to determine the exact predicate. For example, the condition `i<authorList.getLength()` of the `for` loop for the Java component shown in Figure 2a means that the iterator variable `i` is incremented until its value reaches the number of items in the `authorList` object. However, this condition could be `i<getInput()`, where the function `getInput()` receives the number from the user at runtime. In Viola a general approach is taken to abstract away predicate conditions of branching statements. For branching control statements only `loop` and `if-then` statements are used in abstract programs.

7 Symbolic Execution

Symbolic execution is a path-oriented evaluation method that describes data dependencies for a path [28][29][19]. Program variables are represented using symbolic expressions that serve as abstractions for concrete instances of data that these variables may hold. The state of a symbolically executed program includes values of symbolic variables. When a program is executed symbolically its state is changed by evaluating its statements in a sequential order.

7.1 Background

Historically, symbolic evaluation is used for analyzing and testing programs that perform numerical computations. We illustrate it on a simple example. Consider two consecutive statements `x=2*y` and `y=y+x` in a program. Initially, variables `x` and `y` are assigned symbolic values `X` and `Y` respectively. After symbolically executing the first statement, `x` has the value `2*Y`, and after executing the second statement the value of `y` is `Y+2*Y` and the value of `x` remains unchanged. When symbolically executing numerical programs, variables obtain symbolic values of polynomial expressions.

Recall that *symbolic execution trees (SETs)* are graphs characterizing the execution paths followed during the symbolic executions of a program. Nodes in these graphs correspond to executed statements, and edges correspond to transitions between statements. Each node in a symbolic execution tree has a unique identifier. *Path condition (PC)* is a boolean expression over the symbolic variables designating properties that symbolic variables must satisfy in order for an execution to follow some path. Each node describes the current state of execution that includes values, the statement counter, and the PC. PCs for initial nonbranching statements are set to `true`. Nodes for branching statements (e.g., `if` or `while` statements) have two edges that connect to nodes with different PCs.

7.2 Executing Abstract Operations Symbolically

We execute abstract programs symbolically on schemas. Nodes in the SET contain the values of symbolic variables and (modified) schemas. Each abstract operation creates a new node in the SET and updates the content of the symbolic state variables, and in addition, operations `Add` and `Delete` modify schemas. We define the operational semantics of abstract operations in terms of changes made to the state variables and schemas after these operations are executed.

The operation `Load` marks the beginning of abstract programs and instructs Viola to create a new SET, initialize the state variables, and read in the schema for the data. Viola processes abstract operations until the `Save` operation is encountered, which marks the end of the abstract program. The `Navigate` operation, just like the `Load` and `Save` operations does not change the schema, but it modifies the content of the state variables to reflect what elements and attributes are navigated to and what their child elements are. Other operations that do not modify schemas are the operations `Read` and `Write`. When executed they create new nodes in the SET, and update the `operation` state variable with the name of the abstract operation. These operations also update the value of the rw state variable with paths to data elements that are read or written to.

Operations Add and Delete add elements to and delete elements from schemas and update values of state variables to reflect the changes. Recall that elements in schemas have bounds that are specified with attributes minOccurs and maxOccurs. If the Add operation is executed unconditionally, then it gives us assurances that there should be at least one instance of the added element in the data. Correspondingly, the value of the minOccurs attribute is set to one if it was previously set to zero, or left unchanged if it was greater or equal to one. After the Delete operation is executed unconditionally, then there may not be a single instance of the deleted element. Correspondingly, the value of the minOccurs attribute is set to zero to reflect the possibility that there may not be any data instances of a given element.

Conditional execution of the Add and Delete operations occurs when these operations are located within the body of conditional statements (e.g., if-then) or loops. In case of loops, we often lack the knowledge of the upper bound on the number of iterations through the loop, and we assume that it is infinite. A conservative guess of the lower bound on the number of iterations through the loop is zero, meaning that it is never executed. If the Add operation is executed within the body of a conditional statement or a loop, then the entry for the added element is inserted into the schema with the attribute minOccurs set to zero, and the attribute maxOccurs is not present reflecting the absence of the upper bound on the number of instances of this element. If the name of the added data element is not known at compile time, then an edge with the $\diamond$ label is added to the schema graph incident on the current element.

Executing the Delete operation within the body of a conditional statement means that a single instance of the given element may be deleted from data. In general, the name of a deleted element may not be known, and Viola makes a conservative guess about what elements may be affected by these operations. If the Delete operation is conditionally executed within the body of a loop on the set of n elements in the schema and it is not known what elements are deleted, then it is assumed that any subset of the set of these elements can be deleted from the data at runtime. Formally, this is expressed with the powerset operator $\mathcal{P}$ which transforms the set of n elements to 2^n subsets of these elements.

Executing the Add operation creates a new node in the SET. If the element being added already exists in the schema, then the schema is not changed. Otherwise, the schema is modified by adding an entry that describes the added data element as a child to the currently navigated element. The child symbolic variable is updated with the path to the newly added element.

Bookmarking commands "state <name>" and "jump <stateName>" mark certain states in executions of abstract programs and return to them so that the execution resumes in a certain state. When the command "state <name>" is encountered after some abstract operation, the symbolic executor marks the node corresponding to this operation with the name <name> specified as a parameter to this command. When the command "jump <stateName>" is executed, the symbolic executor moves to the node marked with the <stateName> name, which was set by executing some previous command state, and the execution is continued from there.

Three global symbolic path variables are associated with each SET. The access path variable Θ keeps paths to navigated elements and attributes, the delete path variable Δ keeps paths to deleted elements and attributes, and the add path variable Ω keeps paths to added elements and attributes. Each path is associated with a node in a symbolic execution tree. Function `GetSetNodePath: path`$\rightarrow$`SETNodeID` maps a path to the unique identifier of a node, `SETNodeID`, in a SET. For each component C_i and data D_j there is a triple $\langle C_i, D_j, \langle\Theta, \Delta, \Omega\rangle\rangle$ that maps this component and data to access, delete, and add paths of the data elements accessed and modified by this component. Function `GetPaths:` $C \times D \rightarrow \langle\Theta, \Delta, \Omega\rangle$ maps a component C accessing and manipulating data D to the path triple $\langle\Theta, \Delta, \Omega\rangle$.

7.3 Example of Symbolic Execution

Simplified SETs for the abstract programs for the Java and C++ component (shown in Figure 4) are shown in Figure 8. Nodes in the trees correspond to selected operations in the abstract programs. The content of a node includes the schema of the data and symbolic state variables. Due to the lack of space, only schemas are shown in the nodes of the trees. Solid block arrows point to nodes denoted by the state variable `current`. Values for path variables for the execution tree nodes are shown in Tables 2, 4, and 3.

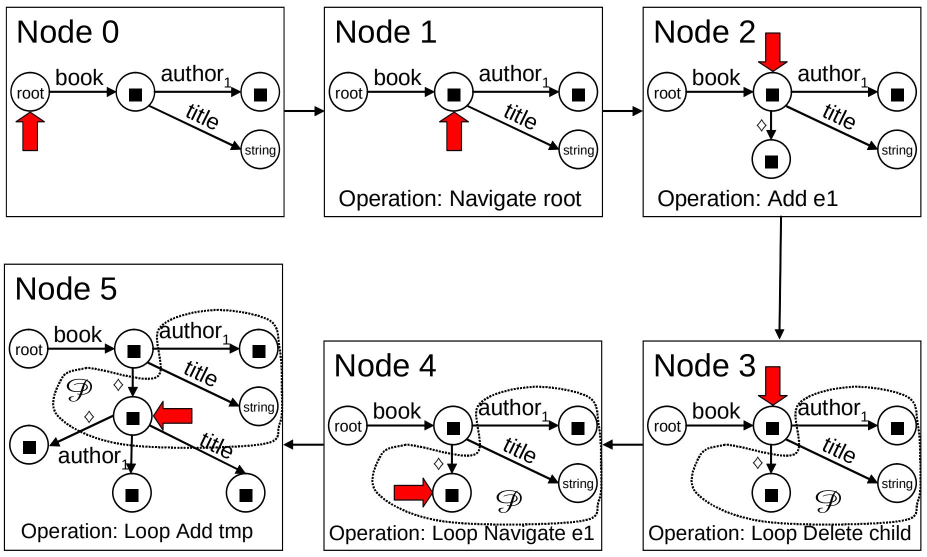
First, we describe the symbolic execution of the abstract program shown in Figure 4a for the Java component shown in Figure 2a on the schema shown in Figure 5a. The symbolic execution tree is shown in Figure 8(a), and the values of the symbolic state variables are shown in Table 2. The `Node 0` shows the initial schema `S` of the data. After executing the abstract operation `Navigate root`, the node `Node 1` is created, and the schema remains unchanged. Access path `book` is added to the access path variable Θ as shown in the table Table 3 in the column for the Java component paths.

The next operation `Add e1` adds the symbolic element `e1` as a child to the currently navigated element book. Correspondingly, the path {`book`, $\diamond$} is added to the add path variable Ω. Since the value for the variable `e1` is not known at compile time, an edge labeled $\diamond$ is added to the schema graph. Node `Node 2` is created, and its variable current remains unchanged, but the content of the variable child is updated with the path to the newly added element `e1`. We do not show the operation `tmp = child` that copies values of the child variable to the `tmp` symbolic variable.

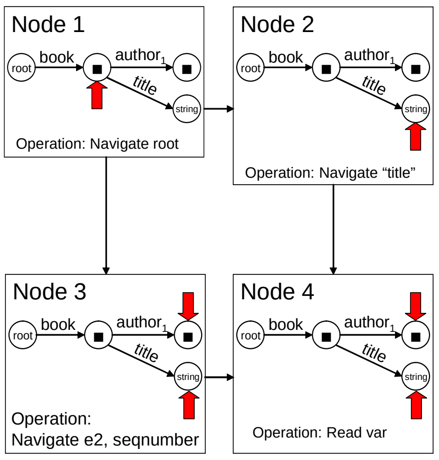
Next, the operation `Delete child` is executed within a loop, and it deletes the children of the element `book`. Since the condition of the loop is not known in the

Node	**Operation**	**$\langle$current, child$\rangle$**
Node 1	Navigate root	$\langle\{\{\text{book}\}, \{\{\text{book, author}\},\{\text{book, title}\}\}\}\rangle$
Node 2	Add e1	$\langle\{\{\text{book}\}, \{\{\text{book, author}\},\{\text{book, title}\},\{\text{book}, \diamond\}\}\}\rangle$
Node 3	Loop Delete child	$\langle\{\{\text{book}\}, \{\{\text{book, author}\},\{\text{book, title}\},\{\text{book}, \diamond\}\}_{\mathcal{P}}\}\rangle$
Node 4	Loop Navigate e1	$\langle\{\{\text{book}, \diamond\}, \{\}\}\rangle$
Node 5	Loop Add tmp	$\langle\{\{\text{book}, \diamond\}, \{\{\text{book}, \diamond, \text{author}\},\{\text{book}, \diamond, \text{title}\},\{\text{book}, \diamond, \diamond\}\}_{\mathcal{P}}\}\rangle$

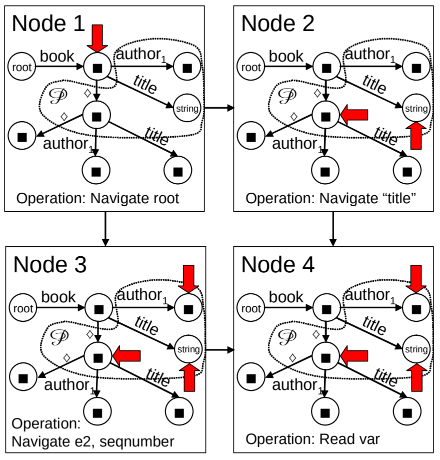
Table 2. Values of the state variables for the symbolic execution tree shown in Figure 8(a).



(a) A simplified symbolic execution tree for the abstract programs shown in Figure 4a and the schema shown in Figure 5a.



(b) A simplified SET for the abstract program shown in Figure 4b on the schema shown in Figure 5a.



(c) A simplified SET for the abstract program shown in Figure 4b on the schema obtained as a result of the symbolic execution shown in Figure 8(a).

Fig. 8. Simplified SETs for the abstract programs shown in Figure 4.

abstract program, it is difficult to predict what children will be deleted, if any at all. Therefore it is assumed that any child of the element book can be deleted. The node `Node 3` is created with the schema remaining unchanged, and the $\mathcal{P}$ operator is applied to the variable `child`, showed in the Table 2 as a subscript to the values of the variable `child`. Correspondingly, the paths {`book`, `author`}, {`book`, `title`}, and {`book`, ◇} are added to the delete path variable Δ as shown in Table 3.

Path vars	Java component paths	C++ component paths	
		Schema S	Schema S'
Access paths Θ	{book}, {book, ◇}	{book}, {book, author}, {book, title}	{book}, {book, author} {book, title}, {book, ◇}
Delete paths Δ	{book, author}, {book, title}, {book, ◇}		
Add paths Ω	{book, ◇}, {book, ◇, author}, {book, ◇ title}, {book, ◇, ◇}		

Table 3. Values of access, delete, and add path variables after symbolically executing abstract programs for Java and C++ components shown in Figure 4.

Within the same loop the operation `Navigate e1` is executed, which is reflected in the `Node 4`. Access path {`book`, ◇} is added to the access path variable Θ. Since the previously stored access path {`book`} is a subpath of the path {`book`, ◇}, i.e., {`book`} ⊆ {`book`, ◇}, the path {`book`, ◇} may be removed from the access path list for optimization. Finally, the operation `Add tmp` appends children of the element book to the currently navigated element `e1`, updating the add path variable Θ with the paths {`book`, ◇}, {`book`, ◇, `author`}, {`book`, ◇, `title`}, and {`book`, ◇, ◇}. We designate the resulting schema shown in the `Node 5` as `S'`.

Node	Operation	⟨current, child⟩
Node 1	Navigate root	⟨{{book}, {{book, author},{book, title}}}⟩
Node 2	Navigate "title"	⟨{{book, title},{}}⟩
Node 3	Navigate e2, seqnumber	⟨{{{book, author},{}}, {{book, title},{}}, {book, ◇}, {book, ◇, ◇}, {{book, ◇, author},{book, ◇, title}}}⟩

Table 4. Values of the state variables for the symbolic execution tree shown in Figure 8(c).

Now we describe the symbolic execution of the abstract program shown in Figure 4b. Programs are executed both on the original schema `S` and on the schema `S'`. Symbolic execution trees for the schemas `S` and `S'` are shown in Figures 8(b) and 8(c) respectively, the values of the symbolic state variables are shown in Table 4, and the values for the path variables are shown in Table 3.

The `Node 1` shows the result of executing the operation `Navigate root` on the schema `S'`. The path {`book`} is added to the access path variable Θ for the C++ component. Executing the conditional statement `if` creates nodes `Node 2` and `Node 3` for operations `Navigate` "`title`" and `Navigate e2` respectively. Access paths {`book`, `author`}, {`book`, `title`}, and {`book`, ◇} are added to the access path variable Θ as shown in the Table 3 in the column for the C++ component paths for the schema `S'`. Correspondingly, only access paths {`book`, `author`} and {`book`, `title`} are added when the program is executed on the schema `S`.

8 Comparing Schemas

Consider the model of interoperating components shown in Figure 1. After executing an abstract program of the component `J` symbolically on the schema describing the XML data $\mathtt{D}_1$, a schema `S'` is obtained that approximates the data $\mathtt{D}_2$ that would be output by this component. Recall the main safety property that is defined as the XML data $\mathtt{D}_2$ should conform to the schema `S`. This property is validated during the verification step of Viola. The goal of this step is to compare the schema `S'` with the schema `S`. If these schemas are equal, then the data instance that conforms to one schema also equally conforms to the other schema, and the components that use this data are compatible. Otherwise, components are not compatible with respect to the data and may throw run-time errors. We describe a *bisimulation* algorithm that is used to compare schemas, and we give an example of its use.

8.1 Bisimulation

Formalization of schemas as graphs is described in Section 3.3. An efficient method to determine if two graphs are equal is bisimulation [11]. A bisimulation is a binary relation between the nodes of two graphs $g_1, g_2 \in \mathtt{G}$, written as $x \sim y$, $x, y \in \mathtt{V}$, satisfying the following properties:

Property 1 if x is the root of g_1 and y is the root of g_2, then $x \sim y$;
Property 2 if $x \sim y$ and one of x or y is the root node in its graph, then the other node is the root node as well;
Property 3 if $x \sim y$ and $\mathtt{type}(x)=\mathtt{type}(y)$ and $x \xrightarrow{r_p^q} x'$ in g_1, then there exists an edge $y \xrightarrow{s_k^m} y'$ in g_2, with the same labels `r=s`, and `max(r)=max(s)` and `min(r)=min(s)`, and $\mathtt{type}(x')=\mathtt{type}(y')$, such that $x' \sim y'$;
Property 4 conversely, if $x \sim y$ and $\mathtt{type}(x)=\mathtt{type}(y)$ and $y \xrightarrow{l_k^m} y'$ in g_2, then there exists an edge $x \xrightarrow{l_p^q} x'$ in g_1, with the same label `l`, and `p=k` and `m=q`, and $\mathtt{type}(x')=\mathtt{type}(y')$, such that $x' \sim y'$.

Two finite graphs $g_1, g_2 \in \mathtt{G}$ are equal is there exists a bisimulation from g_1 to g_2. A graph is always bisimilar to its infinite unfolding. Computing bisimulation of two graphs starts with selecting the root nodes and applying the above properties. When a relation (x, y) between nodes x and y in a graph is found that fails to satisfy the above properties, then the graphs are determined not equal and the bisimulation stops. This relation is called *offending*.

8.2 Example of Bisimulation

We demonstrate how to apply the bisimulation algorithm to show whether two schemas shown in Figure 9a and Figure 9b are equivalent. These schemas describe instances of XML data shown in our motivating example in Figure 2b and Figure 2c. Specifically, a Java component shown in Figure 2a reads in the XML data shown in Figure 2b that conforms to the schema shown in Figure 9a and outputs the modified data shown in Figure 2c that conforms to the schema shown in Figure 9b. A C++ component shown in Figure 2d reads in the data which it expects to conform to the schema shown in Figure 9a. If the schema G_b of the data modified by the Java component (which is shown in Figure 9b) is equivalent to the schema G_a expected by the C++ component (which is shown in Figure 9a), then we can declare both components compatible with respect to the data.

First, we select the root nodes in both schemas which satisfy Property 1 and Property 2. Next, we select relation $\texttt{root} \xrightarrow{\texttt{book}} \blacksquare$ from the schema G_a and check to see that the Property 3 holds for the relation root $\texttt{root} \xrightarrow{\texttt{book}} \blacksquare$ from the schema G_b. Since it does, we determine that the Property 4 holds for both relations, and we proceed to the relation $\blacksquare \xrightarrow{\texttt{author}_1} \blacksquare$ for the schema G_a and the relation $\blacksquare \xrightarrow{\texttt{authors}} \blacksquare$ for the schema G_b. We determine that Property 3 and Property 4 are violated and $\blacksquare \xrightarrow{\texttt{authors}} \blacksquare$ is the offending relation. Thus, these graphs are not equal, and the verification step fails.

9 The Algorithm

Abstraction and verification phases of Viola determine whether the main safety property holds for interoperating components. Recall that the secondary safety property is defined as navigation paths to data elements should not intersect with paths to elements added and deleted by interoperating components. Ensuring that the secondary safety property holds guarantees the absence of PP and PS types of errors. The essence of the refinement step of the Viola algorithm is to guarantee that the secondary safety property holds.

PS and PP errors are hardest to catch, and information about them is the most valuable for diagnostics. Catching PP and PS errors requires a different approach then bisimulation. Consider the basic model shown in Figure 1 when component `J` adds or deletes some elements from, and component `C` accesses some other elements in the data `D`$_2$. When symbolically executed on the schema of the data `D`$_2$, operations of the component `J` modify it so that the bisimulation algorithm will find offending relations and

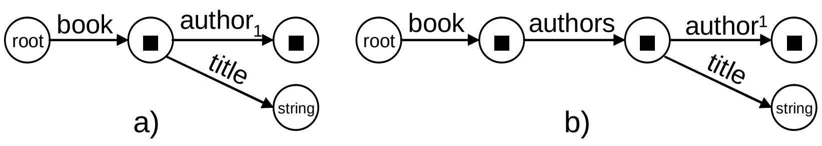


Fig. 9. Graphs for two XML schemas describing the data shown in Figure 2b and Figure 2c.

determine that the schemas S and S′ do not match. However, in general, component J may modify elements that lie on a different path than the elements accessed by the component C. While the mismatch between schemas may lead to errors, no errors will result from the execution of components in this specific case. Therefore our algorithm should be precise in distinguishing between situations that have potential for errors and situations that are highly likely to lead to errors.

Abstract operations Navigate, Read, and Write access data elements in paths without changing them while operations Add and Delete modify paths. By creating lists of accessed and modified paths for interoperating components, it is possible to determine whether they intersect and subsequently, if modifications made by some components may lead to errors in other components. We use this idea in the error catching algorithm ViolaAlgorithm that is shown in Figure 10.

We describe ViolaAlgorithm in whole, not just its refinement stage. The algorithm is initiated after applying component analyses techniques for identifying abstract operations in the components source code. A table (C, SL, EL, FSA) links components to abstract operations identified by the FSAs, where C is the component identifier, SL and EL are the Starting Line and the Ending Line of the component's source code that contain a sequence of API calls that is accepted by the FSA. Another table (C, SL, EL, AP) links a component C to some abstract programs AP. A pair (AP, SET) maps abstract programs AP to SET of these programs. A triple (SET, SETNode, FSA) links the node SETNode in a SET to the abstract operations identified by the FSA. The input to this algorithm is the set of components C. For each pair of distinct components c_i and c_j from the set C, it is determined whether these components interact via the set of data $\{D\}$. This set is obtained by using the function GetData: $C \rightarrow \{D\}$ that maps the component C to the set of data $\{D\}$ that it uses. This function is applied to the components c_i and c_j to obtain the sets of data $\{D\}_i$ and $\{D\}_j$ respectively, and the intersections of these sets of data is computed to determine the common data used by both components c_i and c_j. Then for each element of the computed set of common data, the procedure CatchErrors is called.

Procedure CatchErrors takes identifiers of two components q and t interacting using the data d. After the component q receives the data d that conforms to the schema S_q, it is executed on this data, and passes the modified data to the component t that expects the data to conform to the schema S_t. To obtain schemas S_q and S_t the function GetSchema: $C \times D \rightarrow S_c$ is called to map the component C to the schema S_C that describes the data D. Procedure ComputeSchema symbolically executes the abstract program of the component q on the schema S_q, producing the schema S'_q. Then the procedure Bisimulate is run on the schemas S_t and S'_q in order to determine whether they are equivalent. If these schemas are equivalent, then we terminate the error checking algorithm without reporting any errors. Otherwise, the error reporting the schema mismatch is printed and the procedures Catch-PP-Errors and Catch-PS-Errors are invoked to report PP and PS types of errors in the components q and t. The procedure Catch-PP-Errors is shown to catch PP-1 errors only, since the logic for catching PP-2 errors is similar.

The idea of the Catch-PS-Errors procedure is to check to see that each path in the set of paths accessed by interoperating components, P, exists in the schema graph,

```
**ViolaAlgorithm**( C )
BEGIN
for each $c_i \in C$ do
  for each $c_j \in C \wedge c_i \neq c_j$ do
    GetData($c_i$) $\mapsto \{D\}_i$
    GetData($c_j$) $\mapsto \{D\}_j$
    $D = \{D\}_i \cap \{D\}_j$
    if $D \neq \emptyset$ then
      for each $d \in D$ do
        CatchErrors($c_i$, $c_j$, d)
      end for
    end if
  end for
end for
END
```

```
Catch-PP-Errors( $\Theta$, $\Delta$ )
BEGIN
for each $p_\Theta \in \Theta$ do
  for each $p_\Delta \in \Delta$ do
    if $p_\Delta \subseteq p_\Theta$ then
      print error PP-1: accessing
         deleted data elements
    end if
  end for
end for
END
```

```
Catch-PS-Errors( P, S )
BEGIN
for each $p \in P$ do
  length(p) $\mapsto$ n
  root(S) $\mapsto$ current
  for k = 1 to n do
    GetLabel(p, k) $\mapsto l_k$
      if $l_k = \diamond$ then
        children(current) $\mapsto$ current
      else if $l_k \in$ current then
        children($l_k$) $\mapsto$ current
      else
        print error: nonexistent
                  data elements
      end if
  end for
end for
END
```

```
CatchErrors( $q \in C$, $t \in C$, $d \in D$ )
BEGIN
GetSchema(q, d) $\mapsto S_q$
GetSchema(t, d) $\mapsto S_t$
ComputeSchema(q, $S_q$) $\mapsto S'_q$
if Bisimulate($S_t$,$S'_q$) = false then
  print error: schema mismatch
  GetPaths(q, d) $\mapsto <\Theta_q, \Delta_q, \Omega_q>$
  GetPaths(t, d) $\mapsto <\Theta_t, \Delta_t, \Omega_t>$
  Catch-PP-Errors($\Theta_t$, $\Delta_q$)
  Catch-PS-Errors($\Theta_t$, $S'_q$)
  Catch-PS-Errors($\Theta_q$, $S_q$)
  Catch-PS-Errors($\Theta_t$, $S_t$)
  Catch-PS-Errors($\Omega_q$, $S_t$)
end if
END
```

Fig. 10. Viola algorithm for catching PP and PS errors in interoperating components.

`S`. It means that for each label in the given path the transition with this label should exist in the schema graph. If no such transition is found, then an error is reported to programmers.

The procedure `Catch-PP-Errors` takes sets of access and delete paths, Θ and Δ, as its input and computes whether a delete path is a subpath of some access path. If a delete path is a subpath of some access path, then a possible PP-1 error is reported to programmers.

Going back to the procedure `CatchErrors`, we observe that it invokes the procedure `Catch-PP-Errors` with parameters Θ_t and Δ_q in order to compute if paths in Δ_q are subpaths of Θ_t. Then it calls the procedure `Catch-PS-Errors` four times. First, it determines if the access paths of the component `t`, Θ_t, exist in the computed schema S'_q. Next, it verifies if the access paths of the component `q`, Θ_q, exist in the original schema $\mathtt{S_q}$. Then, the same check is applied the access paths of the component $\mathtt{t}$, $\Theta_{\mathtt{t}}$, and the original schema $\mathtt{S_t}$. Finally, the add paths of the component $\mathtt{q}$, $\Omega_{\mathtt{q}}$, are checked to see if they exist in the schema $\mathtt{S_t}$.

When an error is found, it is important to map it to the source code in order to improve the quality of diagnostics. This mapping is accomplished in a series of steps. First, recall that each path is associated with a node in a SET, and the function `GetSetNodePath` maps a path to the unique identifier of a node in a SET. Each tree is mapped to an abstract program, and a node of the tree is mapped to a specific abstract operation in this program. From the triple (`SET`, `SETNode`, `FSA`), the abstract operations identified by the FSA is obtained. From the pair (`AP`, `SET`) it is determine the abstract programs `AP` whose `SET` is analyzed. From the table (`C`, `SL`, `EL`, `AP`) the component `C` and the its scope are determined that match the abstract programs `AP`. Finally, from the table (`C`, `SL`, `EL`, `FSA`) that links components to abstract operations identified by the FSAs, exact scope of the component source code is determined that leads to the found error.

10 Prototype Implementation

A prototype implementation of the Viola architecture shown in Figure 3 is based on the EDG Java front end C++ and Java parsers [7] and an MS XML parser. FSAs, abstract programs, schemas, SETs, and even output errors are provided in the XML format, and we use the ROOF framework [24] to process XML documents. Our prototype implementation included the analysis routines, symbolic executor, schema comparator (bisimulator), and path analyzer with our error checking algorithms. We wrote these components of Viola in Java and interfaced them with EDG front end parsers written in C++ using the Java Native Interface. Our implementation contains less than 9,000 lines of Java and C++ code.

11 Experimental Evaluation

In this section we describe the methodology and provide the results of experimental evaluation of the Viola on subject programs.

11.1 Subject Programs

We applied Viola to two commercial programs for legal and paper supply chain domains (the latter uses a large XML schema with over `1,600` types), and to open source programs obtained from the Internet. One commercial application was written for a legal office, and it used the `Metalex` schema. `MetaLex` is an open XML standard for the markup of legal sources [26]. The other commercial application was written by a now defunct startup company for `papiNet`, a transaction standard for the paper and forest supply chain [27]. The combined source code of both commercial applications was about `30,000` lines of C++ and Java code.

Open source programs are taken from different XML projects posted on the Internet. The `Book` and `Employees` projects contain programs that generate, access, and manipulate XML data that describe books and employees respectively [2]. Since schemas are not available for these applications, we created them based on the available XML data. `ProbeMsg` is an application that creates XML data and sends it as a stream to a different application [6]. `HomeOwners` applications illustrates the use of the Xerces DOM parser to access and manipulate XML data that contains information on homeowners includes their names, addresses, and closing dates of house purchases [5]. Finally, the `Happycoding` website contains different applications that exchange XML data [4]. Open source programs are small, ranging from less than a hundred to less than a thousand lines of code.

11.2 Methodology

To evaluate Viola, we carry out two experiments to explore how effectively it catches errors in the existing interoperating components, and how the precision of the data flow analysis affects the number of false positives. We inject different bugs in the subject programs and test how Viola catches them. We carried out our experiments using MS Windows XP that ran on Intel Pentium 4 3.2GHz dual CPUs and 2Gb of RAM.

In the first experiment, we run Viola on subject programs with injected bugs. The goal of this experiment is to determine how effective Viola is in catching bugs. We report the sizes of the original programs in *lines of code (LOC)*, number of nodes in the corresponding CFGs, and the number of platform APIs used by programs to access and manipulate XML data. We measure times taken by Viola to produce abstract programs and to run the algorithm to report bugs. The time it takes to catch API errors is included in the time of producing abstract programs because these errors are a part of building program abstraction routines.

We inspect the source code of the subject programs before running Viola, and we modify source code to inject bugs. Thus, we expect Viola to catch a certain number of bugs. Viola may catch more bugs for two reasons. It is possible that we miss some existing bugs during the code inspection, and Viola can report false positives. We report the number of expected bugs, detected bugs, and false positives.

We are also interested in the breakdown of bugs based on their types as described in Section 4. We report the number of expected and found bugs for each type of errors. This information helps us to identify the effectiveness and precision of Viola for different types of problems.

The goal of the second experiment is to evaluate the effect of having precise names of data objects versus symbolic variables in abstract programs on the Viola's rate of false positives. Since program abstractions are approximations of actual programs, actual values are often replaced with symbolic variables. For example, if a real program has a variable whose value is a name of a data object, and the value of this variable is computed at runtime, then Viola uses a symbolic variable with an undefined value. In order to compute precise values for abstract programs we need to use more sophisticated analysis that takes more time and resources. In order to know that this additional effort is justified, we manually replace symbolic variables in abstract programs with precise names of data objects and we measure the number of false positives. The measured number of false positives is a function of the percentage of the using precise data object names versus symbolic variables in abstract programs. If the number of false positives does not decrease, then improving data flow analysis will not lead to increased effectiveness of Viola. On the contrary, if Viola reports fewer false positives with the increased precision of data flow analysis, then our approach is practical, can be improved and used in the industrial settings.

11.3 Results

Experimental results with testing Viola on subject programs are shown in Table 5. This table is divided into four main columns. The first column gives the name of the subject project. Next column, `Size`, contains five subcolumns reporting sizes of subject programs, number of types in XML schemas, number of nodes in CFGs, LOCs of abstract programs, and the numbers of API calls in the subject programs respectively. The third column reports the analysis time in seconds and contains two subcolumns for the time it takes to generate abstract programs and the time to catch errors. Finally, the `Bugs` column reports the number of bugs. Its first subcolumn shows the number of expected bugs for each subject project, the subcolumn `Detected` gives the number of bugs caught by Viola, and the subcolumn `False Positives` shows the number of false positives. For example, the `ProbeMsg` subject program was expected to have `15` bugs, but after running Viola `32` bugs were detected, `14` of which were confirmed through manual code inspection as false positives. It means that three more bugs were missed during the initial code inspection.

A breakdown of expected and detected errors by their types for each subject program is shown in Table 6. The difference between the number of detected and expected errors is the number of false positives. A graphic distribution of the number of detected and expected errors by their types is shown in Figure 11. Almost a half of all false positives are generated by the PP type of errors. Recall that PP errors occur in components that access data elements that are deleted by some other components (PP-1) and by components that read or write wrong elements (PP-2). The reason for a high rate of false positives is that Viola approximates paths through the data during symbolic exe-

Subject Program	Size					Analysis Time, sec		Bugs		
	Programs, LOC	Schema, No. of Types	CFG No. of Nodes	APs, LOC	APIs, No. of calls	Generating APs	Error Detection	Expected	Detected	False Positives
Book	109	16	682	16	27	22.7	3.2	10	16	6
Employees	638	11	2486	30	43	90.5	5.1	15	28	13
ProbeMsg	921	8	7301	57	82	183.6	4.7	15	32	14
Homeowners	147	12	662	32	61	28.3	1.8	15	27	12
Happycoding	372	34	924	47	58	50.4	9.2	15	36	21
papiNet	11048	1653	46934	916	1281	1958.3	28.3	30	103	58
MetaLex	18479	66	63937	835	1526	2739.0	42.9	15	59	46

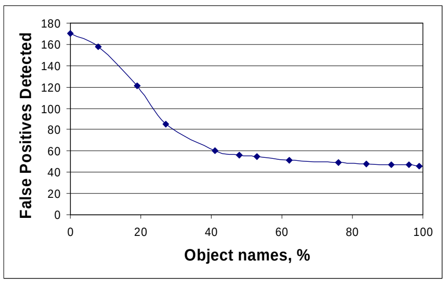
Table 5. Experimental results of testing Viola on commercial and open source projects.

cution. This approximation results in many spurious paths that are not navigated to or modified when components interoperate at runtime.

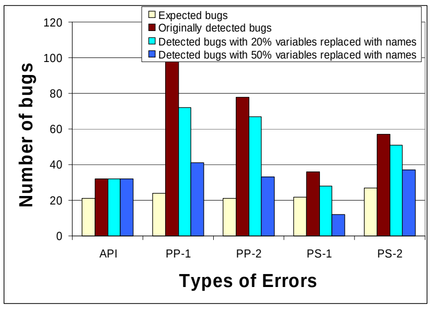
Project	API Errors		PP-1		PP-2		PS-1		PS-1	
	Expected	Found	Expected	Found	Expected	Found	Expected	Found	Expected	Found
Book	2	2	2	5	2	2	2	3	2	4
Employees	3	5	3	6	3	7	3	4	3	6
ProbeMsg	3	4	3	12	3	6	3	3	3	7
Homeowners	3	3	3	8	3	9	3	3	3	4
Happycoding	3	6	3	11	3	10	3	5	3	4
papiNet	6	17	6	25	6	31	6	13	6	17
MetaLex	1	5	4	21	1	13	2	5	7	15

Table 6. A breakdown of real and detected errors by error types.

The results of evaluating the effect of having precise names of data objects versus symbolic variables in abstract programs on the Viola's rate of false positives are shown in Figure 12. The horizontal axis shows the percentage of symbolic variables that we replaced in abstract programs with actual data object names, and the vertical axis shows the number of false positives. We observe that when close to 30% of symbolic variables are replaced with actual object names, the number of false positives decreases approximately three times. Such a significant drop in false positives justifies the use of elaborate data flow analyses that help to improve the precision of the generated abstract programs.



(a) Dependency of false positives issued by Viola from the percentage of data object names versus symbolic variables used in abstract programs.



(b) The distribution of the number of detected and expected errors by their types. The difference between the number of detected and expected errors is the number of false positives.

Fig. 11. Experimental results of testing Viola.

12 Related Work

Our work uses a variety of ideas introduced in different model checkers. Most of these model checkers use the same abstract-verify-refine verification paradigm that Viola is based on. Unlike other model checkers that determine whether programs match specifications or satisfy certain logic predicates (invariants), Viola concentrates on verifying that two components interoperating using XML data do not violate the predefined safety properties. In doing so, Viola employs many common techniques used in other model checkers, but in a novel way.

MAGIC is a model checker that creates models of C components using the method of predicate abstraction, and compositionally verifies computed models using SAT solvers [15]. Like our research, MAGIC uses state machines called linear transition systems to express the desired behavior of systems, and it operates on the source code of C components. By contrast, the Viola's goal is to verify that two components do not violate each other's properties by computing incorrect data.

MOPS is a model checker for verifying that programs do not violate predefined security properties [16]. Like our research, MOPS uses FSAs to describe security properties of programs source code, and it computes models of verified programs by analyzing APIs that affect security properties. Unlike our approach MOPS is used strictly to discover violations of security properties rather than to verify component interoperability.

SLAM is a model checker for C programs that is based on the method of counterexample-driven refinement [14]. SLAM extracts boolean programs from C programs and performs the reachability analysis on the extracted boolean programs by combining interprocedural dataflow analysis and the binary decision diagrams techniques. If a path that leads to an error is not reachable, then SLAM tools analyze the feasibility of paths by discovering additional predicates to refine boolean programs. Like our research, SLAM builds abstract programs and performs path analysis in order to catch errors. Unlike Viola, SLAM does not address verification of interoperating components with respect to the safety properties defined in terms of exchanged data, and SLAM analyzes execution paths in programs, not in the data that they manipulate.

Blast is a model checker for C programs based on a lazy abstraction algorithm. BLAST uses specifications for temporal properties written in C syntax [25]. For model checking Blast uses the predicate abstraction method [23] to find bugs or prove the specification.

Moped is a combined linear temporal logic and reachability checker for pushdown systems [35]. Since pushdown systems can express programs with recursive procedures, its power is equal to or greater than that of Viola's. Moped can also process boolean programs and interact with the SLAM checker.

SLAM, MOPED, and BLAST use the predicate abstraction method [23]. Like our research, these model checkers can verify safety properties of programs using their source code. However, these model checkers are not designed to verify properties of interoperating components that use platform APIs to interact by accessing and modifying data. By contrast, our solution abstracts away properties of interoperating components that are not related to their interacting using data, and it analyzes paths in data computed by symbolically executing abstract programs

A static program analysis method checks structural properties of code by computing an initial abstraction of the code that over-approximates the effect of function calls [36]. Like Viola, this method then refines the computed abstractions by inferring a context-dependent specification for each function call, so that only as much information about a function is used as is necessary to analyze its caller. Rather than concentrating on specifications for function calls, Viola analyzes APIs that access and manipulate XML data.

An automated verification system for XML data manipulation operations translates XML data and XPath expressions to Promela, the input language of the SPIN model checker [22]. The techniques of this system constitute the basis of a web service analysis tool that verifies linear temporal logic properties of composite web services. Unlike Viola, this system cannot be applied to arbitrary C++ and Java programs, however, Viola can use its ideas to further improve the verification process of interoperating components.

13 Discussion and Future Work

Concrete instances of the model shown in Figure 1 are common even in small software systems. Two components that inadvertently modify the same environment variable work fine when running on separate computers, however, they malfunction when put on the same machine. The reason is that the first component sets the value of the environment variable, and this value is not recognized by the other component thus leading to an error that is extremely difficult to catch. Similar situations occur when components use databases, XML and HTML data, or system registries.

The main application of our work is to detect errors when components interact via XML data using XML parsers as underlying platforms. However, components may violate each other's properties by using any kind of data hosted by different platforms. This is known as emergent phenomena in complex systems, specifically software, that occur due to interactions between the components of a system over time. Emergent phenomena are often unexpected, nontrivial results of simple interactions of simple components. Currently, no compiler checks interoperating components for violations that occur as results of these interactions, even when components are located within the same program. We believe that this paper is the first research step in this direction.

Our work has limitations. It is not clear what effort is required in general to create FSAs for different platform APIs. While it is possible to use systems to extract these FSAs from the source code of components [12], it remains to be seen whether automatically extracted FSAs have enough precision to be used in Viola. When verifying C++ components that use pointers, especially function pointers, the number of false positives for API errors is increased significantly. In addition, if no explicit names of data objects are used in components, then the number of false positives would be excessive. However, in our experience once the source of a possible error is located, determining whether it leads to actual errors is easier.

14 Conclusion

We present a novel solution called Viola for verifying safety properties of components interacting via XML data. Viola can detect a situation at compile time when one component modifies XML data so that it becomes incompatible for use by other components. We implemented a prototype of Viola in C++ and Java using EDG Java and C++ and XML parsers. Viola's conservative static analysis mechanism reports potential errors or ensures their absence for a system of interoperating components. We tested Viola on open source and commercial systems, and we detected a number of known and unknown errors in these applications with good precision thus proving the effectiveness of our approach.

Acknowledgments

The author warmly thanks Don Batory for reading this paper and providing valuable comments and suggestions that improved its quality.

References

1. Adobe PDF/XML architecture - working samples. *http://partners.adobe.com/public /developer/en/xml/AdobeXMLFormsSamples.pdf*.
2. The book and employees projects. *http://totheriver.com/learn/xml/xmltutorial.html*.
3. eXtensible Markup Language (XML). *http://www.w3.org/XML/*.
4. The happycoding website. *http://www.java.happycodings.com/XML/index.html*.
5. Homeowners applications. *http://www.sambito.net/AddExampleWeb/navJava.htm*.
6. The probemsg project. *http://www.akadia.com/services/java-xml-parser.html*.
7. Edison Design Group. *http://www.edg.com*.
8. XML Schema. *http://www.w3.org/XML/Schema*.
9. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, January 1991.
10. *Cost Analysis of Inadequate Interoperability in the* U.S. *Capital Facilities Industry, GCR 04-867*. NIST, August 2004.
11. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and* XML. Morgan Kaufmann, October 1999.
12. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
13. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122, 2001.
14. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
15. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.
16. H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, 2002.
17. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

18. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, January 2000.
19. L. A. Clarke and D. J. Richardson, editors. *Symbolic evaluation methods for program analysis.* Prentice-Hall, 1981.
20. M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *ICSE*, pages 177–187, 2001.
21. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000.
22. X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *ISSTA*, pages 252–262, 2004.
23. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.
24. M. Grechanik, D. S. Batory, and D. E. Perry. Design of large-scale polylingual systems. In *ICSE*, pages 357–366, 2004.
25. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.
26. http://www.metalex.nl/pages/welcome.html. *Metalex*, 2002.
27. http://www.papinet.org. *papiNet*, 2002.
28. J. C. King. A program verifier. In *IFIP Congress (1)*, pages 234–249, 1971.
29. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
30. D. Lee, M. Mani, F. Chiu, and W. W. Chu. NeT & CoT: Inferring XML schemas from relational world. In *ICDE*, page 267, 2002.
31. D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.
32. J. Meier, S. Vasireddy, A. Babbar, and A. Mackman. Improving .NET application performance and scalability. *Microsoft Corporation*, 2004.
33. R. Schmelzer. Breaking XML to optimize performance. Z*ap*T*hink* LLC - *special to SearchWebServices.com*, Oct. 2002.
34. D. Spinellis. A critique of the Windows application programming interface. *Computer Standards & Interfaces*, 20(1):1–8, Nov. 1998.
35. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A java bytecode checker based on Moped. In *TACAS*, pages 541–545, 2005.
36. M. Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153, 2004.