Finding Errors in Interoperating Components

Mark Grechanik
Systems Integration Group, Accenture Technology Labs
Chicago, IL 60601
Email: {mark.grechanik}@accenture.com

***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 *Verifier for Interoperating cOmponents for finding Logic fAults (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 compared 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.**

I. 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 Interface (API)* functions 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 API functions that different components invoke 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[1] 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 errors. For example, programmers may use platform API calls incorrectly in the component source code, and modify XML data so that it becomes incompatible for use by other components. Currently, no tool checks interoperating components for potential flaws in their source code that lead to incorrect data exchanges and runtime errors, even when components are located within the same application.

Our solution is a ***V**erifier for **I**nteroperating c**O**mponents for finding **L**ogic f**A**ults (Viola)* that finds errors in components exchanging XML data and helps test engineers to validate reported errors. Viola creates models of the source code of components and computes approximate specifications of the data (i.e., schemas[2]) that these components exchange. The input to Viola is the component's source code, schemas for the XML data used by these components, and *Finite State Automata (FSAs)* that model abstract operations on data with low-level platform API calls. 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 API calls to access and manipulate XML data.

Viola uses control and data flow analyses along with the provided FSAs to extract abstract operations from the component source code. Next, these operations are symbolically executed to compute approximate schemas of the data that would be output by these components. That is, given the schema of the input data, Viola reengineers the approximate schema of the data that would be output by some component from its source code.

The reengineered and expected schemas are compared to determine if they match each other. If a mismatch between them is found, it means that some component modifies the data incorrectly so that runtime exceptions may be thrown by other components using this incorrect data. To confirm this, Viola 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 some potential errors are reported to help test engineers validate and reproduce errors.

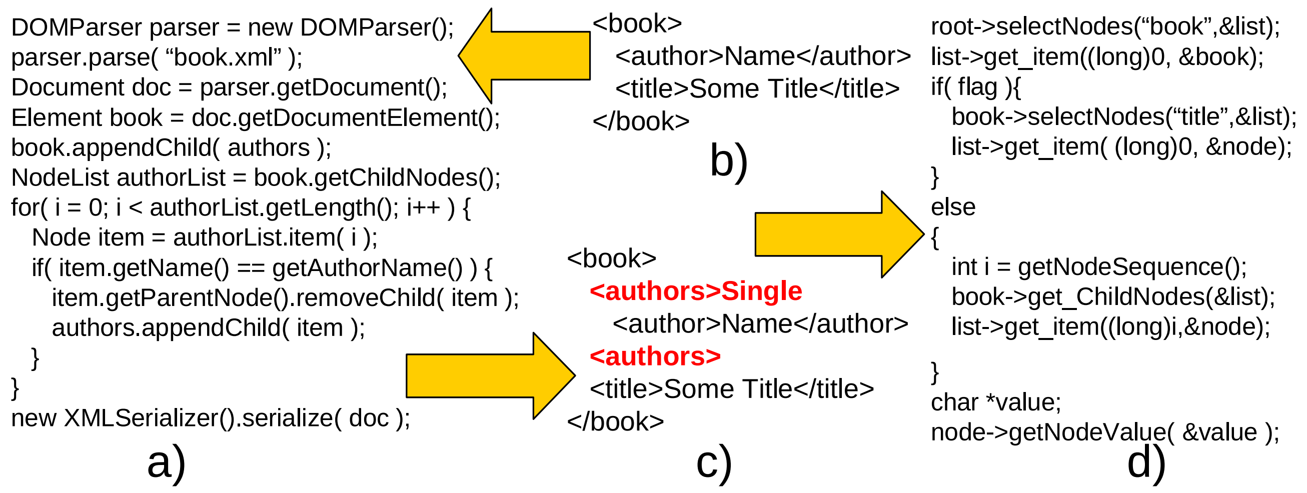
Viola is a helpful bug finding tool whose static analysis mechanism reports some potential errors for a system of interoperating components. We tested Viola on open source and commercial systems, and we detected a number of known and unknown errors in these applications with good precision thus showing the potential of this approach.

II. A Motivating Example

Consider the example shown in Figure 1 as fragments of Java (Figure 1a) and C++ code (Figure 1d) for two respective

[1]A capital facility is a structure or equipment which generally costs at least $10,000 and has a useful life of ten years or more.

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



```
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>Single
  <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
{
  int i = getNodeSequence();
  book->get_ChildNodes(&list);
  list->get_item((long)i,&node);

}
char *value;
node->getNodeValue( &value );
```

Fig. 1. Java (a) and C++ (d) components that interoperate using XML data (b) and (c).

components that interoperate using XML data (Figure 1b-c). 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 that is shown in Figure 1b. This XML data describes the attributes of a book that include the author and title. The Java component modifies the structure of the XML component by adding the tag `authors` as a child element of the root element `book` and moving the `author` element under the tag `authors`. The resulting XML data is shown in Figure 1c.

The C++ component shown in Figure 1d reads in the XML data shown in Figure 1b-c, and depending on the value of the boolean variable `flag`, 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 1b. When the Java component modifies this data, the C++ component may throw a run-time exception because the element `author` is not present in the XML data under the root element `book`.

Currently, there are various projects that address this problem by making XML a first-class data type at the language level (e.g., XJ, XLinq, Xact, and Cω). While some success has been demonstrated, these projects have three major problems. First, they impose additional type systems and new coding practices on programmers, and these additions serve as inhibiting factors for adopting these approaches. Second, for these languages to be sound (i.e., to ensure the absence of bugs if the compiler reports no errors) programmers should not compute names of XML data elements at runtime. This constraint limits programmers to a small class of applications. Third, given the large number of legacy systems that has been written using API calls exported by XML parsers, it is unlikely that these systems will be rewritten any time soon using these approaches.

In our 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 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 API functions are called to access data elements. Either way, runtime errors occur whether XML data is validated or not.

It is possible for a parser to fail validation of XML data against a schema, however, components may never throw runtime exceptions. It happens when different interoperating components do not access and modify the same data elements. For example, the Java component modifies the element `author` by moving it as a child of the inserted element `authors`, and the path to the element `title` remains the same. Even though the XML data shown in Figure 1b and Figure 1c are different, the C++ component will not throw a runtime error when the value of its variable `flag` is `true` returning the title of the book.

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 API calls correctly and translating API calls at compile time into changes that would be made to XML 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 1b. However, if the Java component executes before the C++ component, then it would modify the data into an instance shown in Figure 1c, and thus 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.

III. The Problem Statement

Our goal is create a tool for finding errors in interoperating components that exchange XML data. This tool should report some potential errors when evidence of violating some properties is found. Our approach is neither sound nor complete. A sound approach ensures the absence of errors in components if it reports that no errors exist, and a complete approach reports no errors for correct components.

We use a basic model shown in Figure 2 throughout this paper. In this model, J and C are components (say a Java and C++ components respectively) that interact using XML data D_2. Component J reads in data D_1, modifies it, and passes it as data D_2 to the component C. Component C reads in the data D_2 expecting it to be an instance of some schema S. Since J outputs 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 an instance of a different schema 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 in D_2.

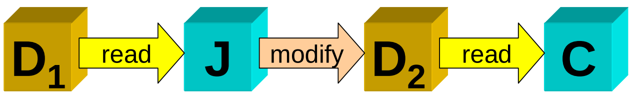


Fig. 2. A model of component interoperability.

There are different reasons why programmers make mistakes when they write the components J and C. Based on our participation in large-scale projects, we observe that programmers often make wrong assumptions about schemas. Given that many industrial schemas contain thousands of elements and types, it is easy to make mistakes about names of elements and their locations in schemas. The other source of errors lies in the complexity of platform API calls that programmers use to access and manipulate XML data. XML parsers export dozens of different API calls, and mastering them requires a steep learning curve.

Often, programmers lack the knowledge of the impact caused by changing the code of some component on other components that interoperate using XML data. This lack of knowledge is an effect of the Curtis' law that states that application and domain knowledge is thinly spread and only one or two team members may possess the full knowledge of a software system [21]. The effect of this law combined with the difficulty of comprehending large-scale XML schemas and high complexity of platform API calls result in components producing XML data that is incompatible for use by other components.

The other source of errors is the disparity in evolving XML schemas and components. Database administrators usually maintain schemas, and programmers maintain components that interoperate using XML data that should be instances of these schemas. If a database administrator modifies some schemas and does not inform all programmers whose components are affected by this change, then some components will keep modifying XML data according to the obsolete schemas.

The problem of mismatch between XML data and schemas is typically addressed by using schema validators that are parts of many XML parsers. In our model shown in Figure 2, an XML parser can validate that the data D_2 is an instance of the schema S when J produces this data. If the data is not an instance of this schema, then the parser 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 schemas for validating XML data is often not attempted because it degrades components performance [37] [36], and it even leads to throwing exceptions when there may not be any runtime errors. Suppose that the component J deletes all instances of some data element thus violating the schema S that requires at least one instance of this element be present in D_2. If either of components J and C validates this incorrect data D_2 against the schema S, then a runtime error will be issued. 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 data elements components J and C access and modify, and if no data element accessed by C is modified by J, then components J and C may still interact safely even if the data D_2 is not an instance of the given schema S.

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. Using API calls exported by XML parsers remains the primary mode of XML access and manipulation. Various language extensions and type systems were proposed to address this problem [17] [33] [26]. Some of these approaches require programmers map XML types to types from the proposed type systems, and that adds complexity to developing interoperating components. Other approaches propose type systems that are not sound or have constraints (e.g., structural modification to XML data are prohibited) that reduce their practicality. To our knowledge these are research approaches, and none of them has been used in commercial or open source projects.

Our goal is to design a tool that does its best to ensure that certain properties hold in components interacting using XML data. These properties are main and secondary safety properties. Given interoperating components J and C producing and exchanging data D_2 at runtime which is an instance of the schema S, the *main safety property (MSP)* is defined as ensuring D_2 conforms to S.

The *secondary safety property (SSP)* is defined as the same data elements in S should not be accessed by one and modified by some other interoperating components provided that specifications are not used at runtime to validate XML data. Since the MSP and SSP ensure stronger guarantee that no runtime will be thrown, using XML parsers to validate data against schemas is irrelevant to our problem. The problem is to find and report some situations at compile time in which interoperating components violate both safety properties. Currently, no tool checks interoperating components for violating these properties, even when components are located within the same application. Viola should output descriptions of execution scenarios that lead to potential errors, and test engineers should be able to follow these scenarios to validate the reported errors.

IV. Errors

We classify errors that Viola catches in interoperating components into the following general categories:

- *Path-Path (P2)* errors occur when a component accesses elements that may be deleted by some other components. P2 errors occur in components that access data elements that are deleted by some other components (P2-1) and by components that read or write wrong elements (P2-2). P2-1 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.
  P2-2 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. Consider a component that reads the value of the first element located under the root "`book`" in the XML data shown in Figure 1b. The read element is "`author`" and the obtained value is "`Name`." However, if the component `J` modified this data as shown in Figure 1c, then the read element would be "`authors`" and the obtained value is "`Single`". Thus, if the component `J` inserts a data element into the path to some elements accessed by the component `C`, then the result of interference of these operations is that the component `C` accesses and reads values of different data elements from what was intended when it uses sequence numbers of elements rather than their names.
- *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 schemas. Suppose that a schema defines the value of the `minOccurs` attribute for a data element to be equal to one, however, a component deletes 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 API calls.
  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 [35] [38]. 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 descriptions of sequences of API calls that expert programmers build for abstract operations, Viola can flag sequences of API calls that do not match any abstract operations as potentially 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 logic faults.

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

P2-1: At line `23` component `C` accesses element $\langle$`book`, `author`$\rangle$ that may be deleted by the component `J` at line `122`.

P2-2: At line `23` component `C` may read a wrong element located under path $\langle$`book`$\rangle$ 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 $\langle$`book`, `author`$\rangle$, however, at least one instance of this element is required by the schema `S`.

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

V. 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 schemas are taken from the model shown in Figure 2. 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 that model abstract operations on XML data `(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 API calls are detected, then API errors are reported to programmers `(4)`.

Running ARs results in abstract programs for `C` and `J` components `(5)`. Abstract programs represent sequences of

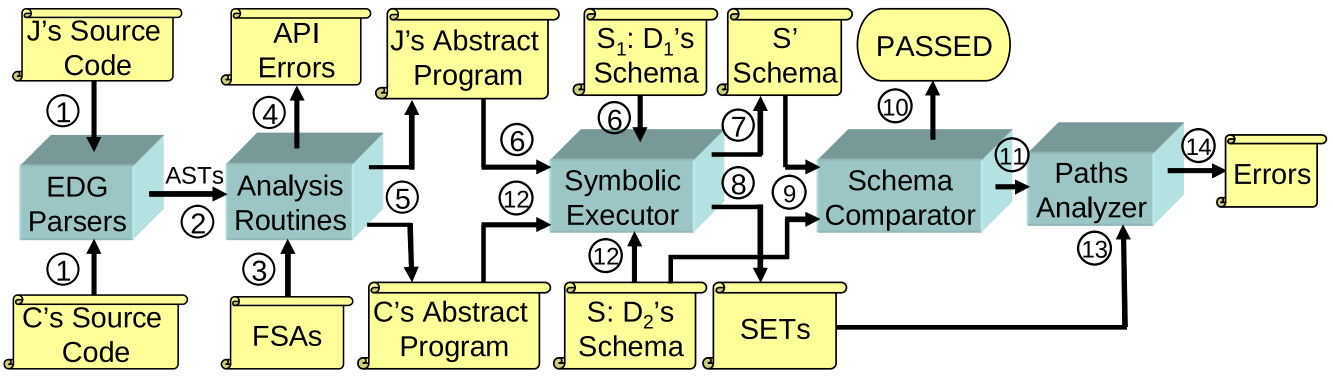


Fig. 3. Viola's architecture and process.

abstract operations on the XML data. The *Symbolic Executor (SE)* executes the abstract program for the component `J` on the schema S_1 of the XML data D_1 `(6)` and outputs the schema `S′ (7)` and *Symbolic Execution Trees (SETs)* `(8)`. 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.

Schema `S′` is the approximate specification of data that would be output by the component `J` if it is executed on the input data D_1. This schema can be viewed as reengineered from the component `J` when its abstract program is symbolically executed on the the schema S_1 of the XML data D_1. This reengineered schema `S′` represents the approximate view of the XML data held by a programmer who wrote the component `J`. Comparing the reengineered schema `S′` with the schema `S` establishes if the MSP is violated, and consequently if the component `J` may perform some incorrect manipulation on the input data.

The *Specification Comparator (SC)* compares the reengineered schema `S′` with the D_2's schema `S (9)`, and reports success if the schemas are the same `(10)`. If this step fails, then the MSP is violated. In the next step `(11)`, Viola checks for violations of the SSP by analyzing if the component `C` accesses data elements that are modified by the component `J`.

To check for the violations of the SSP property, SE executes the abstract program of the component `C` on the schema `S` of the data D_2 `(12)`. The purpose of this step is to obtain information about data elements that the component `C` accesses in the data D_2 provided that it is an instance of the schema `S`. Then SE executes the abstract program of the component `C` on the schema `S′` which is reengineered from the component `J` during the previous steps. The purpose of this step is to obtain information about data elements that the component `C` would access in the data D_2 that is not an instance of the schema S_1, but rather of the reengineered schema `S′`. This information is stored in the SET resulting from this execution, and this SET is added to the set of SETs `(8)`.

The *Paths Analyzer (PA)* analyzes the paths computed by components to accessed and modified data elements `(13)`, and reports the discovered errors to programmers `(14)`. By comparing the paths to elements that the component `C` may access in the data D_2 that is an instance of the schema `S` versus the paths to elements in the data that is an instance of the schema `S′`, PA reports different situations that may lead to P2 errors.

VI. Related Work

Related work on verifying and testing software that accesses and manipulates XML data falls into two major categories: systems that use type checking and verification techniques for XML manipulating programs, and model checkers that automate the verification process for XML-unrelated software artifacts.

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 [23]. 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.

Currently, there are various language design projects that address this problem by making XML a first-class data type at the language level (e.g., XJ, XLinq, Xact, and Cω) [26] [33] [17]. While some success is demonstrated, there are three major problems with these projects. First, they impose additional type systems and coding practices on programmers, and it serves as an inhibiting factor for adopting these approaches. Next, for these approaches to be sound (i.e., to ensure the absence of bugs if the compiler reports no errors) programmers should not compute names of XML data elements at runtime. This constraint limits programmers to a small class of applications. Finally, given the large number of legacy systems that has been written and are being written using API calls exported by XML parsers, it is unlikely that these systems will be rewritten adhering to some of these approaches.

Generator-based approaches (e.g. JAXB, Apigen, Castor) can do automatic mapping for individual languages. For example, if an XML schema contains thousands of types then thousands of corresponding classes are generated in a host programming language that map to these XML types. This approach leads to serious problems with evolution and maintenance of generated code, like a complex naming mechanism, and results in a significantly increased compilation time of the system. Various generators are used as part of programming environments and as standalone tools to generate corresponding types in programming languages. Most are generators that take XML schemas and generate corresponding classes in Java and C++ [11] [12] [13]. This approach requires sophisticated name management software.

Our work uses a variety of ideas introduced in different model checkers [18] [19] [16] [27] [39]. 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.

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 [40]. 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 API calls that access and manipulate XML data.

VII. Conclusion

We present a novel solution called Viola for finding bugs in components interacting via XML data. Viola is a helpful bug finding and testing tool that assists test engineers by detecting 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 static analysis mechanism reports some potential errors 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.

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] *Institute for Software Research, University of California, Irvine, x*ADL *2.0 project,* A*pigen for x*A*rch schemas,.* http://www.isr.uci.edu/projects/xarchuci/tools-apigen.html, 2004.

[12] *Sun Microsystems, Java Architecture for XML Binding (JAXB),.* http://java.sun.com/xml/jaxb, 2004.

[13] *Castor XML databinding framework,.* http://www.castor.org/xml-framework.html, 2005.

[14] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and* XML. Morgan Kaufmann, October 1999.

[15] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[16] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[17] G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in c*mega*. In *ECOOP*, pages 287–311, 2005.

[18] 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.

[19] 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.

[20] L. A. Clarke and D. J. Richardson, editors. *Symbolic evaluation methods for program analysis.* Prentice-Hall, 1981.

[21] B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, 1988.

[22] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *CAV*, pages 232–247, 2000.

[23] X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *ISSTA*, pages 252–262, 2004.

[24] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

[25] M. Grechanik, D. S. Batory, and D. E. Perry. Design of large-scale polylingual systems. In *ICSE*, pages 357–366, 2004.

[26] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. Xj: facilitating xml processing in java. In *WWW*, pages 278–287, 2005.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.

[28] http://www.metalex.nl/pages/welcome.html. *Metalex*, 2002.

[29] http://www.papinet.org. *papiNet*, 2002.

[30] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000.

[31] J. C. King. A program verifier. In *IFIP Congress (1)*, pages 234–249, 1971.

[32] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[33] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Trans. Software Eng.*, 30(3):181–192, 2004.

[34] D. Lee, M. Mani, F. Chiu, and W. W. Chu. NeT & CoT: Inferring XML schemas from relational world. In *ICDE*, page 267, 2002.

[35] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.

[36] J. Meier, S. Vasireddy, A. Babbar, and A. Mackman. Improving .NET application performance and scalability. *Microsoft Corporation*, 2004.

[37] R. Schmelzer. Breaking XML to optimize performance. Z*ap*T*hink* LLC - *special to SearchWebServices.com*, Oct. 2002.

[38] D. Spinellis. A critique of the Windows application programming interface. *Computer Standards & Interfaces*, 20(1):1–8, Nov. 1998.

[39] D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A java bytecode checker based on Moped. In *TACAS*, pages 541–545, 2005.

[40] M. Taghdiri. Inferring specifications to detect errors in code. In *ASE*, pages 144–153, 2004.