

Finding Errors in Components That Exchange XML Data

Mark Grechanik

Accenture Technology Labs
Chicago, IL 60601, USA
mark.grechanik@accenture.com

ABSTRACT

Two or more components (e.g., objects, modules, or programs) interoperate when they exchange data, such as XML data. Using *Application Programming Interface (API)* calls exported by XML parsers remains a primary mode of accessing and manipulating XML, and these API calls lead to various run-time errors in components that exchange XML data. Currently, no tool checks the source code of interoperating components for potential flaws caused by third-party API calls that lead to incorrect XML data exchanges and runtime errors, even when components are located within the same application.

Our solution combines program abstraction and symbolic execution in order to reengineer the approximate schema of XML data that would be output by a component. This schema is compared using bisimulation with the schema of XML data that is expected by some other components. We describe our approach and give our error checking algorithm. We implemented our approach in a tool that we used on open source and commercial systems and discovered errors that were not detected during their design and testing.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging; D.5.12 [Software Engineering]: Interoperability

General Terms

Validation, Experimentation, Algorithms

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 Interface (API)* calls through which components invoke platform services to access and manipulate data. For example, an *eXtensible Markup Language (XML)* parser is a platform for XML data; it exports API calls that different components invoke to access and manipulate XML documents.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'07, Atlanta, Georgia, November 5-9, 2007

Copyright 2007 ACM 978-1-59593-882-4/07/0011 ...\$5.00.

Two or more components interoperate when they exchange information [1]. It is conservatively estimated that the cost of programming errors in component interoperability just in the capital facilities industry¹ in the U.S. alone is \$16 Bil per year. A primary driver for this high cost is fixing flaws in incorrect data exchanges between interoperating components [2].

Although it is known in advance that components exchange data, it is not clear how to detect operations at compile time that lead to runtime errors. Using API calls exported by XML parsers remains a primary mode of accessing and manipulating XML, and these API calls lead to various run-time errors in components that exchange XML data. For example, programmers may use platform API calls incorrectly in the component source code thereby modifying XML data so that it becomes incompatible for use by other components. Currently, no tool checks interoperating components for potential flaws caused by third-party API calls in their source code that lead to incorrect XML data exchanges and runtime errors, even when components are located within the same application.

There are two fundamental problems with using API calls to access and manipulate XML data. First, these API calls take names and types of XML data elements as string input parameter variables. The values of these input parameters are often known only at runtime, making it impossible to apply sound type checking algorithms. Second, XML parsers export dozens of different API calls, and high complexity of these API calls makes it difficult for programmers to understand which API calls to use and how to combine them to access and manipulate XML data [18]. These problems lead to different bugs in components, many of which are difficult to detect at compile time.

Our main contribution is a *Verifier for Interoperating cOmponents for finding Logic fAults (Viola)* that finds errors in components that exchange XML data, and helps test personnel to validate reported errors. Viola builds abstract programs from the source code of components that exchange XML data, and then it symbolically executes these abstract programs on XML schemas 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.

Viola is a helpful bug finding tool whose static analysis mechanism reports certain classes of errors for systems 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 our approach. We also show that by providing names of data elements accessed by API calls, it is possible to reduce the number of reported false positives by more than three times.

¹A 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. THE PROBLEM

Our goal is to find errors in interoperating components that exchange XML data with a high degree of automation and precision. In this section, we describe a model for interoperating components that is used throughout this paper, describe sources of errors and classify them, formulate properties against which we check components, and provide the problem statement.

2.1 Model

We use a basic model shown in Figure 1 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 schema² S_{D_2} . Since J outputs data D_2 before C accesses it, concurrency is not relevant. However, because of design or programming errors, the component J may output 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_{D_2} , a runtime error may be issued when C reads in D_2 .

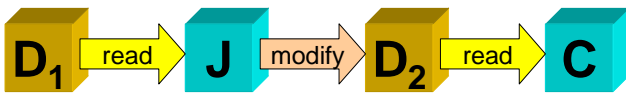


Figure 1: A model of component interoperability.

2.2 Limitations of Schema Validators

The problem of mismatch between XML data and schemas is typically addressed by using schema validators that are embedded components of many XML parsers. Schema validators check XML data with respect to schemas that describe this data. If a mismatch is found, validators throw exceptions.

Unfortunately, schema validators cannot be used to detect errors at compile time for components that use API calls to manipulate XML data. The reason is that the computed XML data that should be checked against their corresponding schemas is not known before components are run. Exceptions will be thrown during this runtime validation either in the component J after it modified the data, or in the component C before it reads the data.

If XML data is not validated at runtime and it is not an instance of its corresponding schema, exceptions will be thrown when certain API calls are invoked to access elements of this XML data. Either way, runtime errors occur regardless from validating XML data against its schemas or not. 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 [21][20], and it even leads to throwing exceptions when there may not be any runtime errors. That is, it is possible for an XML parser to fail validation of XML data against a schema; however, components may never throw runtime exceptions when accessing this data.

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 components J and C validate this incorrect data D_2 against the schema S , then a runtime error will be issued. However, when executed, the component C

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

may never attempt to access the deleted data element, and therefore, no exception will be thrown if the validation step is bypassed.

2.3 Sources Of Errors

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 strength schemas contain thousands of elements and types, it is easy to make mistakes about the names of these 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 hundreds 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 [11]. 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. Business analysts and system 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 without informing all programmers whose components are affected by this change, then some components will keep modifying XML data according to the obsolete schemas.

2.4 What Bugs To Catch

We classify errors that should be caught with our approach 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). P2-2 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 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-1), or when components violate bounds set by schemas on data elements as a result of executing operations on data (PS-2).

PS-2 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.

- *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 [19][22]. One of common mistakes is that programmers use incorrect APIs in the sequences of calls designed to perform operations on data.

2.5 Examples of Error Messages

Below are examples of warnings that will be issued to programmers after using our approach to analyze interoperating components:

P2-1: At line 23 component C accesses element $\langle \text{book}, \text{author} \rangle$ that have been deleted by the component J at line 122.

P2-2: At line 23 component C may read a wrong element located under path $\langle \text{book} \rangle$ because component J modifies elements under this path at line 122.

PS-1: At line 23 component C accesses element $\langle \text{book}, \text{ISBN} \rangle$, however, this element is not defined by the schema S.

PS-2: At line 23 component C may delete all instances of the element $\langle \text{book}, \text{author} \rangle$, however, at least one instance of this element is required by the schema S.

2.6 Safety Properties

When certain properties hold in components that exchange XML data, these components are free of some bugs. These properties are *main and secondary safety properties*. Given interoperating components J and C producing and exchanging XML data D_2 at runtime (as shown in Figure 1) which is an instance of the schema S_{D_2} , the *main safety property (MSP)* ensures that D_2 conforms to the schema S_{D_2} .

It is equally 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_{D_2} . The *secondary safety property (SSP)* is defined as the same data elements in D_2 should not be accessed by one and modified by some other interoperating components.

2.7 The Problem Statement

Our goal is to design a tool that does its best to ensure that the MSP and SSP properties hold in components interacting using XML data. The problem is to find and report some situations at compile time in which interoperating components violate both MSP and SSP properties. Currently, no tool checks interoperating components for violating these properties, even when components are located within the same application.

We assume that two XML schemas exist: the schema S_{D_1} that describes the XML data D_1 which is the input to the component J, and the schema S_{D_2} that describes the expected output XML data D_2 that the component J outputs. A key step is to reengineer a schema S' that approximates the data D_2 from the source code of the component J.

We do not attempt to make our approach either sound or complete. A sound approach ensures the absence of errors in components if it reports that no errors exist and all reported errors are correct, and a complete approach reports all errors or no errors for correct components. Viola should detect certain classes of errors with high automation and good precision, and it should output descriptions of execution scenarios that lead to potential errors, and

test personnel should be able to follow these scenarios to validate the reported errors. When checking the MSP and SSP we assume that schemas are not used at runtime to validate XML data.

3. SOLUTION

In this section, we present core ideas behind our approach, give a high-level overview of how our solution is used to find errors in components that exchange XML data, and present the system's architecture. We use the model shown in Figure 1.

3.1 Abstract Operations

Our main idea is to abstract operations that can be performed on XML data down to seven basic operations: navigating to data elements, reading and writing their values, adding and deleting elements, and loading and saving XML data, designated as *Navigate*, *Read*, *Write*, *Add*, *Delete*, *Load*, and *Save* respectively. Given any XML parser that exports API calls, one or more of these calls are required to implement each of these abstract operations. These abstract operations are implemented in components using low-level platform API calls. It is rare that a one-to-one correspondence exists between abstract operations and API calls. Programmers have to execute sequences of API calls in order to accomplish each of these abstract operations [19][22].

Program abstractions represent source code of interoperating components using abstract operations. We observe that a majority of statements and operations in applications are irrelevant to accessing and modifying XML data. Viola abstracts away specifics of API calls and program variables that do not affect XML data, and transforms applications into sequences of abstract operations. Program abstractions are obtained from source code by analyzing it and mapping sequences of API calls to the abstract operations. An example of a program abstraction for a Java component is shown in Figure 2. With program abstractions we lose precision, however, the number of program states is significantly reduced.

3.2 A Birds-Eye View

A core idea of our approach is to reengineer a schema S' that approximates the data D_2 from the source code of the component J. This reengineered schema S' represents the approximate view of the XML data held by a programmer who wrote the component J. By comparing the reengineered schema S' with the expected schema S_{D_2} we establish if the MSP is violated, and consequently if the component J may perform some incorrect manipulation on the input data D_1 .

The illustration of steps for checking MSP and SSP properties is shown in Figure 3. The first step is to check the MSP property, and the second step is to check the SSP property. Both steps involve creating abstract programs AP_J and AP_C from the components J and C respectively.

The step of checking the MSP property is shown in Figure 3(a). Given the schema S_{D_1} that describes the XML data D_1 which is the

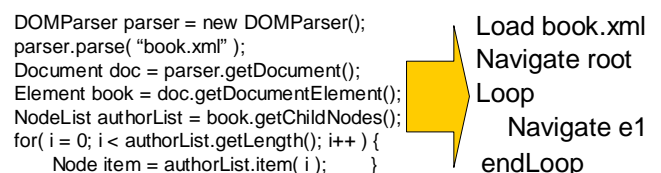


Figure 2: Abstract program (right from the block arrow) extracted from a fragment of Java code (left from the arrow).

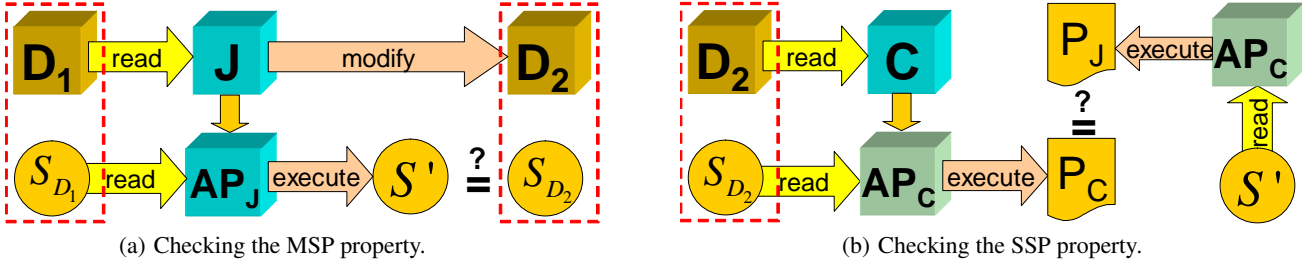


Figure 3: Illustrations of steps for checking MSP and SSP properties.

input to the component J , and given the schema S_{D_2} that describes the expected output XML data D_2 that the component J outputs, the schema S' that approximates the data D_2 is computed by symbolically executing the code of the component J on the schema S_{D_1} .

We give background on symbolic execution in Section 4.2. At this point we define symbolic execution in Viola's context as modifying schemas by applying abstract operations to the elements of these schemas. For example, if the abstract operation `Delete` is symbolically executed in a loop on some XML element, then this element and all its children will be marked as potentially deleted from the corresponding schema.

Once the approximation schema S' is obtained, it is compared with the expected schema S_{D_2} that describes the expected output XML data D_2 . If these schemas are the same, then the MSP is satisfied and no errors are reported.

If these schemas are different, then additional comparison is performed to check to see if the SSP is violated. This step is shown in Figure 3(b). Specifically, it is possible to determine if SSP is violated by analyzing paths to data elements that are accessed by one and modified by other components. To do that, the code for the component C is symbolically executed on the schema S_{D_2} . As a result of this execution, paths P_C are obtained to data elements that are accessed by the component C . These paths are compared with the paths P_J to modified data elements that are obtained as a result of symbolically executing the code of the component J on the schema S_{D_1} . If P_J is a subpath of any path P_C , i.e., $P_C \subseteq P_J$ the violation of the SSP is reported.

3.3 The Architecture of Viola

Viola's architecture and process are shown in Figure 4. 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 1.

The input to the architecture is the J 's and C 's components source code (1). Using language parsers, the source code of the components is parsed into *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 predefined sequences of API calls that model abstract operations on XML data (3), and ARs locate these sequences of API calls in the ASTs of these components. If sequences of API calls that have not been previously defined are detected, then API errors are reported (4). Running ARs results in abstract programs AP_J and AP_C from the components J and C respectively (5).

In order to reengineer a schema S' that approximates the data D_2 from the source code of the component J , the *Symbolic Executor (SE)* executes the abstract program for the component J on the

schema S_{D_1} . Once executed, SE 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.

In order to verify the MSP property, the *Specification Comparator (SC)* compares the reengineered schema S' with the expected schema S_{D_2} (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 symbolically executes the abstract program of the component C on the expected schema S_{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 this schema. Then SE executes the abstract program of the component C on the schema S' (13), 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 may be incorrectly modified by the component J . 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 (14), and reports the discovered errors to programmers (15). 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_{D_2} 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.

To summarize, API and some of PS-2 errors are detected during extracting abstract programs (APs) and symbolic execution respectively. Errors of type PS-1 are detected when comparing the reengineered schema S' and the expected schema S_{D_2} , and errors of type P2 are caught during the path analysis stage of Viola.

4. IMPLEMENTATION OF VIOLA

In this section we describe how abstract programs are obtained from the source code of components, how obtained abstract programs are symbolically executed on XML schemas, and how information about errors is obtained by comparing schemas and analyzing paths to data elements.

4.1 Extracting Abstract Programs

Program abstractions can be obtained from the source code of programs using *Finite State Automata (FSAs)* that map sequences of API calls that access and manipulate XML data to abstract op-

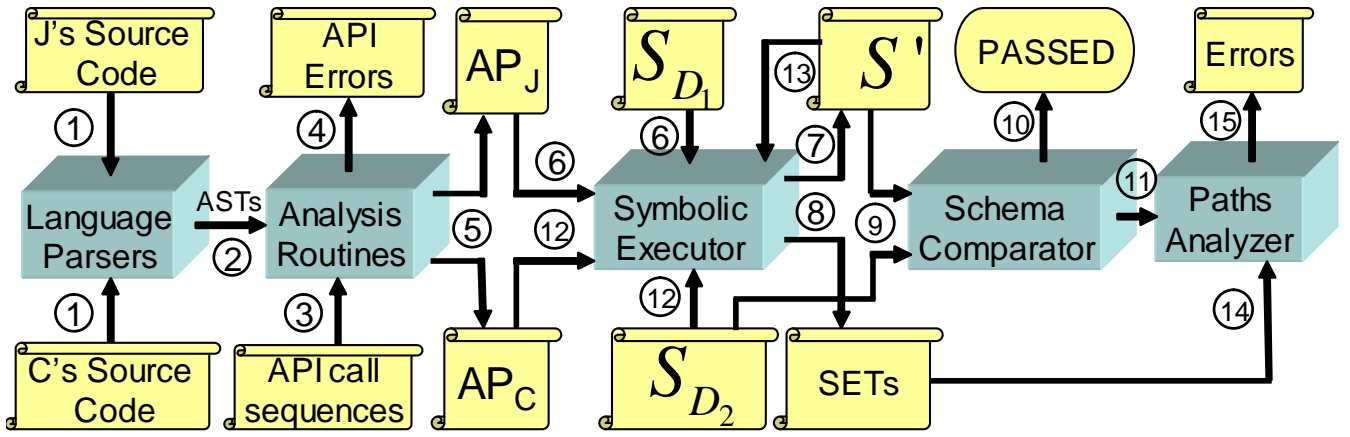


Figure 4: Viola's architecture and process.

erations. Program abstractions are computed for each path in the *Control Flow Graph (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 node in the CFG that contains an API call in the given path, this API call is extracted and put on the stack. Every time an API call is added, the sequence of API calls on the stack is checked to see if it 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.

An example of an FSA for accepting API calls exported by a Xerces XML parser for the abstract operation `Load` is shown in Figure 5. 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 except for the edges incident on the start nodes. 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 can be 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. If a function pointer is used to reference an API call, then Viola is unable to resolve it without the guidance from users. To understand the limitations of Viola, 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 requires guidance from programmers to resolve object references.

In general, it is not possible to determine from the components source code how many times loops will execute. This lack of knowledge is reflected by abstracting away conditional predicates of loops, leaving only information about the loop itself in abstract programs. Thus, the `Loop` and `endLoop` keywords are put into the abstract program.

The lack of knowledge about the navigated data element is reflected by using symbolic variable as parameters to abstract operations. For example, in the abstract program shown in Figure 2

the variable `e1` is used as a parameter to the abstract operation `Navigate`. The values of these variables are the names of the actual data elements. In general, it is an undecidable problem to retrieve names of data elements using static program analysis. However, when abstract operations are executed on schemas, symbolic variables take specific values as parameters to these operations. We show this process in Section 4.2.

4.2 Symbolic Execution

Symbolic execution is a path-oriented evaluation method that describes data dependencies for a path [15][16][10]. 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 the sequential order.

4.2.1 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 $Y+2*Y$. When symbolically executing numerical programs, variables obtain symbolic values of polynomial expressions.

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 the SET describes the current state of execution that includes values of symbolic variables and the statement counter. Nodes for branching statements (e.g., `if` or `while` statements) have two edges that connect to nodes with different condition predicates.

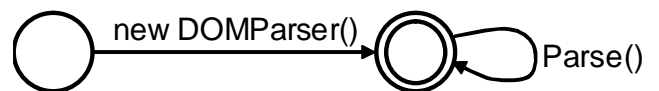


Figure 5: Example of an FSA for accepting API calls exported by a Xerces XML parser for the abstract operation `Load`.

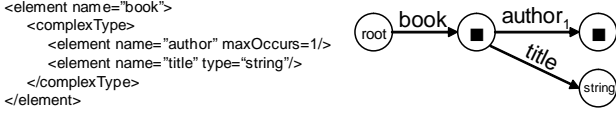


Figure 6: Representing a schema as a graph.

4.2.2 Schemas

Abstract programs are executed symbolically on schemas producing SETs. Recall that nodes in SETs 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. We define the operational semantics of abstract operations in terms of changes made to the state variables and schemas after these operations are executed.

Each schema has a root specified with the `<schema>` element. Each data element is defined by its name and its type. Elements can be either of simple or complex types. 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, the values of elements may be constrained through enumerating or specifying low and upper bounds for numerical types. The second constraint 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. We use these bounds in symbolic execution to determine if components violate them.

We represent schemas using labeled and directed graphs, and we use this formalism for executing abstract programs on schemas symbolically and for comparing different schemas in order to detect discrepancies that lead to runtime errors. An example of a graph for a schema is shown in Figure 6.

Let T be the finite set 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

- $V \subseteq T$, the nodes are type names or \blacksquare if the type of data is not known;
- $L \subseteq F$, 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 ;
- Bounds for elements are specified with subscripts and superscripts to labels designating these elements. Subscripts are used to specify bounds defined by the `minOccurs` attribute, and superscripts designate the bounds specified by the `maxOccurs` attribute;
- Each graph has a special node labeled `root` $\in V$, where `root` represents a collection of the root elements. An empty schema has a single root node and no edges. element is not known.

A path in a schema graph $G = (V, E, L)$ is defined as a sequence of labels $p_G = \langle l_1, l_2, \dots, l_n \rangle$, 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.

Path p_i is a subpath of some other path, p_j , $p_i \subseteq p_j$ if and only if all labels of the path p_i are also labels of the path p_j , and the order

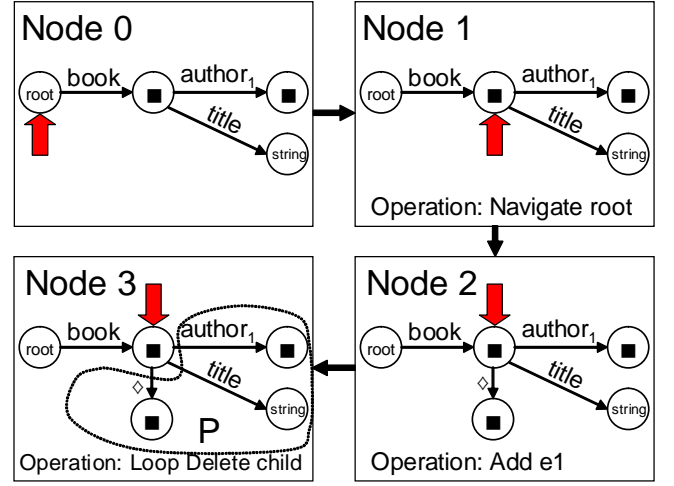


Figure 7: A SET for a symbolically executed abstract program.

in which they appear in p_j is the same as they appear in the path p_i .

4.2.3 Example of Symbolic Execution

We demonstrate an example of symbolic execution of abstract programs on schemas using a graphic representation of a SET as shown in Figure 7.

The Node 0 shows the initial schema for input data D_1 . After executing the abstract operation `Navigate root`, the node Node 1 is created, and the schema remains unchanged because this operation does not modify the schema. The SET stores the information that the path `(book)` is accessed by this abstract operation.

The next operation `Add e1` adds an element denoted by the symbolic variable `e1` as a child to the currently navigated element `book`. Node Node 2 is created. Since the value for the variable `e1` is not known at compile time, an edge labeled \diamond is added to the schema graph and the path `(book, \diamond)` is stored in the SET.

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 abstract program, it is difficult to predict what children will be deleted, if any at all. Therefore it is assumed that any subset of children of the element `book` can be deleted. That is, the children elements are tagged as potentially deleted. Correspondingly, the paths `(book, author)`, `(book, title)`, and `(book, \diamond)` are marked in the SET as potentially deleted by this operation.

In general it is an undecidable problem to determine automatically how many times to execute symbolically abstract operations located within the body of a loop. We decided to make it a configurable parameter in Viola based on the *small scope hypothesis* [14] stating that in practice, many bugs can be detected in small scopes. By default, Viola executes the bodies of loops once.

4.3 Checking the MSP and SSP

Recall the MSP states that the XML data D_2 should be an instance of the schema S_{D_2} . This property is validated by comparing the schema S' with the schema S_{D_2} . 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 runtime errors.

Formalization of schemas as graphs is described in Section 3.3. An efficient method to determine if two graphs are equal is bisimulation [6]. We define *bisimulation* as a binary relation between the nodes of two graphs $g_1, g_2 \in \mathcal{G}$, written as $x \sim y$, $x, y \in \mathcal{V}$, 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 $\text{type}(x) = \text{type}(y)$, and nodes x, y are not tagged as potentially deleted, and $x \xrightarrow{r_p} x'$ in g_1 , then there exists an edge $y \xrightarrow{s_k} y'$ in g_2 , with the same labels $r=s$, $r \neq \diamond$ and $s \neq \diamond$, and $\max(r) = \max(s)$ and $\min(r) = \min(s)$, and $\text{type}(x') = \text{type}(y')$, such that $x' \sim y'$;

Property 4 conversely, if $x \sim y$ and $\text{type}(x) = \text{type}(y)$, and nodes x, y are not tagged as potentially deleted, and $y \xrightarrow{l_k} y'$ in g_2 , then there exists an edge $x \xrightarrow{p_q} x'$ in g_1 , with the same label l , $l \neq \diamond$, and $p=q$ and $m=q$, and $\text{type}(x') = \text{type}(y')$, such that $x' \sim y'$.

Function $\text{type}: \mathcal{V} \rightarrow \mathcal{S}$ returns the type $s \in \mathcal{T}$ of the node $v \in \mathcal{V}$. Function $\max: \text{label}_l^u \rightarrow u$ returns the upper bound u , or ∞ if the upper bound is not specified, and function $\min: \text{label}_l^l \rightarrow l$ returns the lower bound l , or zero if the lower bound is not specified.

Two finite graphs $g_1, g_2 \in \mathcal{G}$ are equal if 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*.

If an offending relation is found, then the MSP property is violated and the SSP property should be verified. This is accomplished by checking if paths modified by the component J are subpaths of paths to elements accessed by the component C , i.e., $P_J \subseteq P_C$.

5. EXPERIMENTAL EVALUATION

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

5.1 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` schema³. `MetaLex` is an open XML standard for the markup of legal sources. The other commercial application was written for `papinet`, a transaction standard for the paper and forest supply chain⁴. 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 respectively⁵. `ProbeMsg` is an application that creates XML data and sends it as a stream to a

³<http://www.metalex.nl>

⁴<http://www.papinet.org>

⁵<http://totheriver.com/learn/xml/xmltutorial.html>

different application⁶. `HomeOwners` applications illustrates the use of the Xerces DOM parser to access and manipulate XML data that contains information on homeowners including their names, addresses, and closing dates of house purchases⁷. Finally, the `Happycoding` website contains different applications that exchange XML data⁸. Open source programs are small, ranging from less than a hundred to less than a thousand lines of code.

5.2 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. To determine how effectively Viola finds bugs, we inject different bugs in the subject programs and test how Viola catches them. We carried out our experiments using Windows XP that ran on Intel Pentium 4 3.2GHz dual CPUs and 2Gb of RAM.

5.2.1 Injecting Bugs

We wish to evaluate how effective Viola is in catching bugs from all categories of errors which are described in Section 2.3. Even though the first attempt to run Viola on subject programs yielded a few faults, they were only API and P2-2 type faults. To test Viola on other types of bugs, we asked several graduate computer science students, each with at least two years of C++ and Java programming experience to insert bugs into subject programs. Initially, these students were not aware about our study and only one student knew how to use Xerces XML DOM parser API calls. After giving instructions on using Xerces and MSXML DOM parser API calls and explaining the goals of this project, each student was asked to inject certain types of bugs in the source code of components. Students were asked to make these bugs as realistic as possible based on their experience and the knowledge of the subject programs. It was up to students to select locations in the source code for the injected bugs and create test cases to ensure that these bugs behave as intended.

When delivered, the subject programs were tested without the injected bugs (the code that students wrote to inject bugs was initially commented out), and then with injected bugs. We excluded injected bugs that introduced compounded bugs in the logic of the existing code accessing and manipulating XML data. That is, if the code for an injected bug led to one or more other bugs, then the injected bug was excluded. The number of bugs remaining in the components source code after exclusions is reported in Table 1.

5.2.2 Experiments

We ran two experiments with Viola. In the first experiment, we ran 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 measured times taken by Viola to produce abstract programs and to run the algorithm to report bugs. The time it took 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 also measured how much memory Viola consumed.

We inspected the source code of the subject programs before running Viola, and we modified source code to inject bugs. Thus, we

⁶<http://www.akadia.com/services/java-xml-parser.html>

⁷<http://www.sambito.net/AddExampleWeb/navJava.htm>

⁸<http://www.java.happycodings.com/XML/index.html>

Subject Program	Size						Analysis Time, sec		Memory, Mb	Bugs		
	Programs, LOC	Schema, No. of Types	CFG No. of Nodes	APs, LOC	APIs, No. of calls	No. of Components	Generating APs	Error Detection		Expected	Detected	False Positives
Book	109	16	682	16	27	3	22.7	3.2	281	10	16	6
Employees	638	11	2486	30	43	8	90.5	5.1	652	15	28	13
ProbeMsg	921	8	7301	57	82	11	183.6	4.7	794	15	32	14
Homeowners	147	12	662	32	61	4	28.3	1.8	335	15	27	12
Happycoding	372	34	924	47	58	3	50.4	9.2	487	15	36	21
papiNet	11048	1653	46934	916	1281	17	1958.3	28.3	1396	30	103	58
MetaLex	18479	66	63937	835	1526	12	2739.0	42.9	1621	15	59	42

Table 1: Experimental results of testing Viola on commercial and open source projects.

expected Viola to catch a certain number of bugs. Viola may report 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 2.3. We report the number of expected and found bugs for each type of errors.

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 Viola’s rate of false positives. Since program abstractions are approximations of actual programs, symbolic variables are often used in place of names of data elements. For example, if a 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 to denote this data element.

Consider a fragment of C++ code shown in Figure 8. The value of the variable `element` is computed at runtime and it is the name of the XML data elements accessed by the `selectNodes` and `get_item` API calls. In general, it is an undecidable problem to determine the values of string expressions at compile time. When extracting an abstract program the variable `element` is replaced with some symbolic variable whose value is undetermined. However, in the C++ code fragment, API calls are executed when the condition `element == "authors"` is met. It means that in the corresponding abstract program the value `authors` can be used in place of the symbolic variable.

```
string element;
int index = 0;
.....
if( element == "authors" ) {
    bookNode->selectNodes(element, &list);
    list->get_item((long)index, &node );
}
```

Figure 8: A fragment of C++ code accessing a data element using DOM API calls.

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. To determine the names of data elements we inspected the source code manually and ran programs collecting values for

variables. After determining names for data elements, we went to abstract programs generated by Viola and manually replaced the corresponding symbolic variables with names of data elements. We symbolically execute abstract programs before and after we replace symbolic variables with data element names, and we measure the number of false positives detected by Viola.

The goal of this experiment is to plot the number of false positives as a function of the percentage of precise names of data element versus symbolic variables denoting these elements in abstract programs. If the number of false positives does not decrease, then improving data flow analysis to detect precise names of data elements will not lead to increased effectiveness of Viola. On the contrary, if Viola reports fewer false positives, the increased precision and subsequently the cost of data flow analysis is justified to improve our approach.

5.2.3 Threats to Validity

Our subject programs are of small to moderate size because we did not have access to large-scale application with components interacting using XML data. Large applications that interact using XML data might have different characteristics compared to our small to medium size subject programs. Increasing the size of applications to millions of lines of code may lead to a nonlinear increase in the analysis time and space demand for Viola. If this happens, then more time should be spent on making Viola scalable.

An additional threat to validity is how faults are injected into the subject programs. The process of fault injection is subjective and poorly controlled in our experimental setup. Specifically, it was difficult to control error propagation through different channel between interoperating components.

5.3 Results

Experimental results with testing Viola on subject programs are shown in Table 1. This table is divided into five 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. The following column show the maximum memory consumption when running Viola. 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`

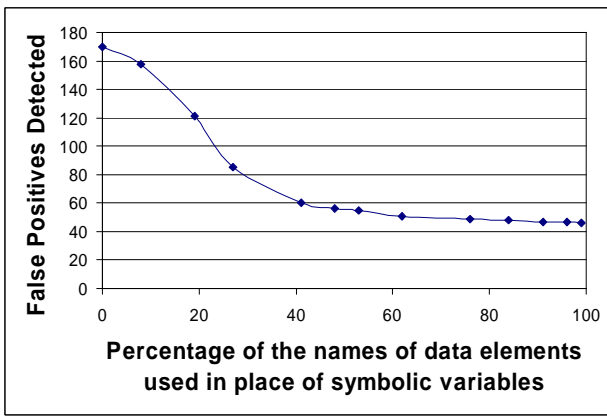


Figure 9: Dependency of false positives issued by Viola from the percentage of precise names of data elements versus symbolic variables used in abstract programs.

subject program was expected to have 15 bugs, but after running Viola, thirty two bugs were detected, fourteen 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 2. 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 10. Almost a half of all false positives are generated by the P2 type of errors. Recall that 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). The reason for the false positives is that Viola approximates paths through the data during symbolic execution. This approximation results in many spurious paths that are not accessed or modified when components interoperate at runtime.

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 9. 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.

5.4 Recommendation

As our experiments show, Viola takes around forty five minutes to run on a system with over 20,000 lines of code. Since programmers routinely run compilers on the code they write many times a day, using Viola this way is prohibitive since it takes too much time to check the code. We suggest that Viola should be used after “freezing” a release and before testing starts. This way test engineers obtain warnings about potential bugs, and they can define test plans to validate these warnings.

5.5 Limitations

Our work has limitations. It is not clear what effort is required in general to create FSAs for different platform APIs. In practice,

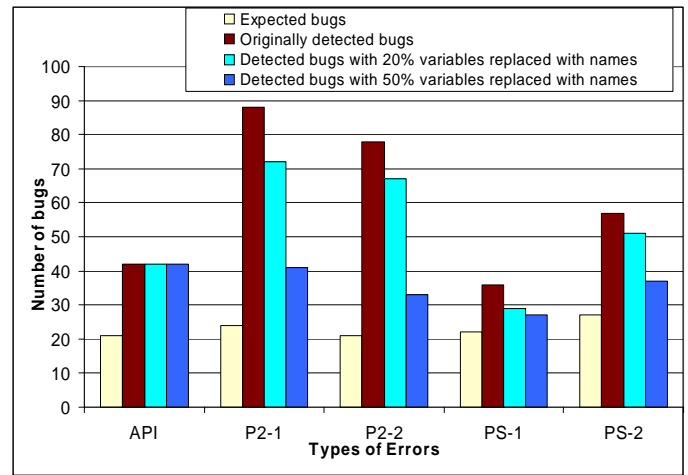


Figure 10: The distribution of the number of detected and expected errors by their types. The difference between the number of detected and expected errors is the number of false positives.

creating FSAs for Xerces and MS XML parser APIs took us less than three days. While it is possible to use systems to extract these FSAs from the source code of components [7], it remains to be seen whether automatically extracted FSAs have enough precision to be used in Viola. When error checking 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, it is easy for programmers to determine whether it leads to actual faults.

6. RELATED WORK

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 [12]. 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, XLink, Xact, and Cω) [13][17][8]. 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 com-

Project	API Errors		P2-1		P2-2		PS-1		PS-2	
	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 2: A breakdown of real and detected errors by error types.

plex 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++ [3][4][5]. This approach requires sophisticated name management software.

Our work uses a variety of ideas introduced in different model checkers. MOPS is a model checker for verifying that programs do not violate predefined security properties [9]. Like our research, MOPS uses FSAs to describe security properties of programs source code, and it computes models of verified programs by analyzing API calls that affect security properties. Unlike our approach MOPS is used strictly to discover violations of security properties rather than to verify component interoperability.

7. 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. Viola's static analysis mechanism reports potential errors for a system of inter-operating 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 and anonymous reviewers for reading this paper and providing valuable comments.

8. REFERENCES

- [1] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, January 1991.
- [2] *Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry, GCR 04-867*. NIST, August 2004.
- [3] *Institute for Software Research, University of California, Irvine, xADL 2.0 project, Apigen for xArch schemas*,. <http://www.isr.uci.edu/projects/xarchuci/tools-apigen.html>, 2004.
- [4] *Sun Microsystems, Java Architecture for XML Binding (JAXB)*,. <http://java.sun.com/xml/jaxb>, 2004.
- [5] *Castor XML databinding framework*,. <http://www.castor.org/xml-framework.html>, 2005.
- [6] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, October 1999.
- [7] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
- [8] G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in *cmega*. In *ECOOP*, pages 287–311, 2005.
- [9] 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.
- [10] L. A. Clarke and D. J. Richardson, editors. *Symbolic evaluation methods for program analysis*. Prentice-Hall, 1981.
- [11] 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.
- [12] X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *ISSTA*, pages 252–262, 2004.
- [13] 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.
- [14] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25, 2000.
- [15] J. C. King. A program verifier. In *IFIP Congress (1)*, pages 234–249, 1971.
- [16] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
- [17] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Trans. Software Eng.*, 30(3):181–192, 2004.
- [18] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *VL/HCC*, pages 199–206, 2004.
- [19] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, pages 48–61, 2005.
- [20] J. Meier, S. Vasireddy, A. Babbar, and A. Mackman. Improving .NET application performance and scalability. *Microsoft Corporation*, 2004.
- [21] R. Schmelzer. Breaking XML to optimize performance. *ZapThink LLC - special to SearchWebServices.com*, Oct. 2002.
- [22] D. Spinellis. A critique of the Windows application programming interface. *Computer Standards & Interfaces*, 20(1):1–8, Nov. 1998.