Automatic Validation of XQuery Programs[*]

Jesús M. Almendros-Jiménez
Dept. of Informatics
University of Almería
jalmen@ual.es

Antonio Becerra-Terón
Dept. of Informatics
University of Almería
abecerra@ual.es

ABSTRACT

In this paper we present a tool for the automatic validation of XQuery programs. Firstly, the tool is able to detect wrong paths in XQuery expressions with respect to an XML Schema. Secondly, it makes possible the specification of input and output properties, as well as input-output properties (i.e., properties relating input and output data) of programs. Thirdly, the tool is able to filter randomly generated test cases with input properties, as well as to test output and input-output properties on randomly generated test cases and the corresponding output. It reports counterexamples when output or input-output properties are not satisfied. The tool has been implemented as an XQuery library which can be used from any XQuery interpreter.

1. INTRODUCTION

From the early years of computing, formal specification of properties has been used to describe the behavior of programs. Properties can describe the input of the program, establishing relationships between input data. Input properties are required for each input, in order to ensure a correct output. Properties can also describe the output of the program, but in this case, they establish the relationships between output data, which are ensured by the program. In this context, testing of programs involves to test input properties for input data as well as to test output properties for each output data. Traditionally, this task is manually performed by the programmer, but in modern programming environments, automatic testing is carried out [2].

Recently [1], we have presented a tool for XQuery enabling the automatic testing of XQuery programs. The tool randomly generates a test suite to test an XQuery program. The test suite is generated from an XML Schema provided by the programmer, in which he/she specifies the elements required in the XML test cases. The tool is implemented as an oracle able to report whether the XQuery expression passes the test. The test is an output property, and thus when it is passed all the outputs of the test cases satisfy the output property. Also the tool reports the number of test cases used for testing. In case of success, the tool reports "Ok". Otherwise the tool reports counterexamples (i.e., test cases of the XQuery program for which the output does not satisfy the given output property). The tool is customizable in the following sense. The human tester can define the structure and content of the test cases from the XML schema. Additionally, the human tester can control the number of test cases. He or she can tune the number and size of XML trees generated from the XML schema. Thus, although the tool generates random test cases, the tester can control the size of the test suite.

In spite of the tool is able to detect many programming errors in XQuery, we have found that some cases require a more elaborated testing process. More concretely, we have found the following drawbacks:

- Most of programming errors in XQuery come from the use of wrong paths in input XML documents. The XQuery programmer tends to make mistakes when he/she specifies the paths on input XML documents. It has as consequence in most of cases empty/wrong answers. However, the structure of XML documents is usually known. Usually, the programmer has an schema (DTD or XML Schema) describing the structure of input XML documents. Thus, a validation of paths in XQuery programs against the schema is required. Moreover, wrong paths make impossible in some cases the testing with our tool, because test cases are generated from the XML Schema. Thus, an XML Schema based validation of paths is mandatory as first step for testing.

- While XML Schema describes the structure of input XML documents, some programs require a more sophisticated characterization of input data. XML Schema enables the specification of the hierarchical structure of XML documents (i.e., the tag structure) as well as cardinality and type of the tags. But there exist some constraints which cannot be specified by the XML Schema. And these constraints can take part of properties required to input data. In our tool test cases are randomly generated but the test case generation algorithm only follows the structure of the XML

[*]This work was supported by the EU (FEDER) and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-44742-C4-4-R, as well as by the Andalusian Regional Government under Project P10-TIC-6114.

iiWAS '15, December 11-13, 2015, Brussels, Belgium
© 2015 ACM. ISBN 978-1-4503-3491-4/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2837185.2837263

Schema. A specification of richer constraints on input data, as well as validation of these input properties/-constraints on input data are required to cover a larger number of programs.

- While validation of input and output properties can be sufficient for some programs, a more sophisticated validation can be carried out specifying relationships between input and output data. In fact, in some cases input (about input data) and output (about output data) properties are not sufficient to describe the behavior of programs. Thus, specification of input-output properties is required.

Here we propose an extension of our tool. Firstly, the extension is able to detect wrong paths in XQuery expressions with respect to an XML Schema. With this aim, the tool analyzes variable dependences and collects the paths on an XQuery expression. Later, it analyzes the conformance of the collected paths to the XML Schema. It reports whether a wrong path has been found. Secondly, the extension of the tool makes possible the specification of input properties, as well as input-output properties. Thirdly, the tool is able to filter randomly generated test cases with input properties, as well as to test input-output properties on randomly generated test cases and the corresponding output. The tool reports counterexamples of input documents when input-output properties are not satisfied.

The properties to be tested are defined by Boolean XQuery functions. Usually, the property expresses a constraint/assert in terms of XPath expressions and logical connectors: for all, exists, and, or, etc., possibly making use of XQuery functions. It makes possible to express a rich repertory of properties: the occurrence of a certain value, the range of a certain attribute, the number of nodes, etc.

Our approach is inspired by similar tools in functional languages. This is the case of the *Quickcheck* tool [4] for Haskell, and the *PropEr* tool [11] for Erlang. Properties in these approaches are specified by functions, and they are automatically tested from random test cases. Since XQuery is also a functional language, it seems natural to provide a similar tool for XQuery programs. Nevertheless, the context is different. XQuery is a functional language handling XML documents which are, in essence, ordered labeled trees, and XQuery has as main element XPath expressions. Thus, here random test case generation is focused on trees and thus specific algorithms can be defined to automatically generate trees of a certain size. Due to the higher order capabilities of XQuery properties, as well as the XQuery program, can be passed as arguments to the tester enabling the implementation of the tool in a similar way to Haskell and Erlang, that is, the tool is an XQuery program implemented as an XQuery library and thus can be used from any interpreter.

Schema based validation of XPath has been studied in some works (see [7, 3, 6, 9], among others). However, as far as we know, schema based validation of XQuery has not been previously studied, and none of the existing XQuery interpreters are equipped with XPath validation from XML Schemas. We have studied how to analyze variable dependences on XQuery programs in order to collect paths. Once paths have been collected an algorithm for path validation is defined. Both procedures have been implemented as an XQuery library.

```
path :=   step | step[cond] | step[/path]
cond :=   path | cond and cond | cond or cond
step :=   axis | axis/step | axis//step
axis :=   n | * | @n | @* | . | .. | text()
```

Figure 1: XPath Syntax

In [5] they propose the automatic generation of XML documents from a DTD and an example. The framework called *GxBE (Generate-XML-By-Example)* uses a declarative syntax based on XPath to describe properties of the input documents. They are able to express global properties on the document, in particular, to express the so-called count constraints making use of XPath and the count function. With regard to GxBE, our approach randomly generates test cases from the XML Schema and after they are filtered by input properties, enabling as particular case the same kind of constraints as GxBE. We will show an example using count constraints. Additionally, we are also able to handle output and input-output properties. In GxBE they are concerned about test case generation efficiency. In our approach, test case generation efficiency cannot be ensured but it is under control. Firstly, the human tester can customize the number of test cases by modifying the input XML Schema, as well as by selecting the size of the trees. Additionally, test case generation is dynamic. It means that when testing a given program, test cases are incrementally generated and the tool stops when a failure (i.e., the property is not satisfied) has been found. It drastically reduces the time required for testing.

2. VALIDATION OF PATHS IN XQUERY

Firstly, we would like to explain how path validation of XQuery is carried out. Given that the tester uses the XML Schema for generating test cases, the same XML Schema is used for path validation. Path validation includes path recollection of the XQuery expression and XML Schema based validation. Path recollection involves variable dependence analysis. Due to the lack of space, we omit here the details about the path recollection of the XQuery expression. However, we show how a path expression is validated with respect to an XML schema.

The collected path expressions are defined according to grammar of Figure 1. Even though the XQuery expression can include paths with references to values, for instance, "book[@year>2000 and price<100]/title", the path recollection remove values, thus becoming "book[@year and price]/title", since we are interested in the validation of the path structure with respect to the XML Schema. While XML Schemas offer a wide range of mechanisms to express input data structure, testing and validation processes only use a small subset of mechanisms. In path validation we use the basic elements of XML schemas, that is, complex types, elements and attributes. Cardinality constraints are not used in path validation but used for test case generation (see [1] for more details about test case generation). We only consider the case of *sequence*, and *ref* can be used to define recursive definitions in which a recursive *complexType* has to be defined. *minOccurs*, *maxOccurs* and *use* elements as well as *enumeration* are used for test case generation but not for path validation. In summary, an XML Schema for test case generation and validation includes: *attribute* (*use*), *element* (*minOccurs/maxOccurs*), *sequence* (*minOccurs/maxOccurs*), *simpleType*, *complexType*, *ref* and *enumeration.* Thus, XML Schemas are defined according to the

xschema :=	ss cs element
complextype :=	ct n els atts
simpletype :=	st n ens
element :=	elem n cardinality | ref n cardinality | elem n complextype cardinality
attribute :=	att n use u
enumeration :=	val n
cs :=	() | complextype | complextype cs
ss :=	() | simpletype | simpletype ss
els :=	() | element | element els
atts :=	() | attribute | attribute atts
ens :=	() | enumeration | enumeration ens
cardinality :=	minOccurs b maxOccurs b

Figure 2: XML Schema Syntax

```
Boolean validator($\bar{c}$,ac,p,xs)
{
if empty(p) then true
else if head(p)=/text() then true
        else if empty($\bar{c}$) and head(p)=/n
                then false
        else if empty($\bar{c}$) and head(p)=/*
                then false
else if head(p)=/* then
        validator($\cup_{1 \leq i \leq n}$ cts($c_i$),$\bar{c}$+ac,next(p),xs)
else if head(p)=/text() then true
else if head(p)=n then
        if name($c_i$)=n then
                validator(cts($c_i$),$\bar{c}$+ac,next(p),xs)
                and validator_pred(cts($c_i$),$\bar{c}$+ac,pred(p),xs)
        else if ref($c_i$)=n then
                validator(ct(xs,n),$\bar{c}$+ac,next(p),xs)
                and validator_pred(ct(xs,n),$\bar{c}$+ac,pred(p),xs)
        else false
else if head(p)=// then
        validator_rec($\bar{c}$,ac,next(p),xs)
else if head(p)=. then
        validator($\bar{c}$,ac,next(p),xs)
        and validator_pred(cts($\bar{c}$),$\bar{c}$+ac, pred(p),xs)
else if head(p)=.. then
        if empty(ac) then false
                else validator(head(ac),tail(ac),next(p),xs)
                and validator_pred(head(ac),tail(ac),pred(p),xs)
else if head(p)=@u then
        if $u \in$ att($c_i$) or u=* then true else false
}
```

Figure 3: XPath validator algorithm

abstract grammar of Figure 2, where n is a string and b is a natural number or "unbounded" and u is "optional" or "required".

The validation algorithm is as follows (see Figure 3). The algorithm takes as input a sequence of complex types $\bar{c} = c_1, \ldots, c_n$, a path p, a sequence ac of ascendants of $\bar{c}$ (i.e., a sequence of $\bar{c}$'s), and an XML Schema xs. It returns *true* when the XPath expression p is validated with regard to xs, otherwise it returns *false*. The initial call to the algorithm is as follows: validator*(ct root el (),(),p,xs)* whenever $xs = ss\ cs\ el$, where root is a fictitious name for the parent of the root element. Given a path p, we denote by $head(p)$ the left most element of p, and $next(p)$ the path in which $head(p)$ has been removed. $Pred(p)$ denotes q when $p = r[q]$. Given a complex type $c = ct\ n\ \bar{e}\ \bar{a}$, we denote by $cts(c)$, the elements c_i, $1 \leq i \leq n$, where $e_i = elem\ n_i\ c_i$, and by $att(c)$ the elements n_i, $1 \leq i \leq n$, where $att_i = att\ n_i$. Given an schema xs we denote by $ct(xs, n)$ the complex type of name n of xs. "+" denotes the concatenation of sequences, and head and tail the head and the tail of a sequence, respectively.

The algorithm distinguishes cases depending on the form of p, and $\bar{c}$. The algorithm keeps stored the ascendants ac of $\bar{c}$ (i.e., the ascendants in the XML Schema of $\bar{c}$) in order to be able to go backward when ".." is used. Also, the schema xs is passed as argument in order to go to the root when "/" is used in the beginning of path conditions. Path conditions are validated by the auxiliary Boolean function **validator_pred**. Also **validator_rec** is an auxiliary function to validate path expressions starting from "//". Due to the lack of space we omit here details about these functions.

The path validator has been implemented in XQuery and it uses the XQueryX [10] representation of XQuery programs. Thus, before path validation, the query is transformed into the corresponding XQueryX representation. Next, an XQuery function is responsible of the traversal of the XQuery code as well as the path validation.

3. TESTING PROCESS

Now, we would like to explain how testing process is carried out. We will define the meaning of a certain program passing a test of input/output properties. We restrict our presentation to programs with unary inputs. The following definitions can be generalized to the case of n-ary inputs.

Assuming a finite set of test cases $t_1, \ldots, t_n$ of an XML Schema Σ, a program $\mathcal{P}$ and a Boolean unary input property p, and a Boolean unary output property q we say that $\mathcal{P}$ passes the test (p, q) in $t_1, \ldots, t_n$ whenever for each t_i, $i = 1, \ldots, n$ such that $p(t_i)$ and $\mathcal{P}(t_i) \neq \emptyset$ then $q(\mathcal{P}(t_i))$. We say that $\mathcal{P}$ fails the test (p, q) whenever there exists t_i, $i = 1, \ldots, n$ such that $p(t_i)$, $\mathcal{P}(t_i) \neq \emptyset$ and $\neg q(\mathcal{P}(t_i))$. Finally, we say that (p, q) cannot be checked for $\mathcal{P}$ in $t_1, \ldots, t_n$, whenever for all t_i, $i = 1, \ldots, n$, $\neg p(t_i)$ or $\mathcal{P}(t_i) = \emptyset$.

Obviously, if (p, q) cannot be checked for $\mathcal{P}$ then the test cases $t_1, \ldots, t_n$ are not sufficient and thus we cannot say anything about the program $\mathcal{P}$. In order to solve this situation the human tester has to modify Σ, p or q. The implemented tester is able to detect (for a finite set of test cases) when a program $\mathcal{P}$ passes or fails a test (p, q) as well as when (p, q) cannot be checked for $\mathcal{P}$. We will show in Section 4 examples of these cases.

Analogously, we can define the following concept. Given a finite set of test cases $t_1, \ldots, t_n$ of an XML Schema Σ, a program $\mathcal{P}$ and a binary Boolean property u we say that $\mathcal{P}$ passes the test u in $t_1, \ldots, t_n$ whenever for each t_i, $i = 1, \ldots, n$ such that $\mathcal{P}(t_i) \neq \emptyset$ then $u(t_i, \mathcal{P}(t_i))$. We say that $\mathcal{P}$ fails the test u whenever there exists t_i, $i = 1, \ldots, n$ such that $\mathcal{P}(t_i) \neq \emptyset$ and $\neg u(t_i, \mathcal{P}(t_i))$. Finally, we say that u cannot be checked for $\mathcal{P}$ in $t_1, \ldots, t_n$, whenever for all t_i, $i = 1, \ldots, n$, $\mathcal{P}(t_i) = \emptyset$.

The implemented tester is also able to detect when a program $\mathcal{P}$ passes or fails a test u as well as when u cannot be checked for $\mathcal{P}$. We will show in Section 4 examples of these cases.

4. EXAMPLES

Now, we show examples of validation of paths with respect to XML Schemas and examples of validation of input/output properties.

Example 1 Assuming the schema of Figure 4 we can validate with the path validator the conformance of the following program:

```
for $t in (
for $book in $bookstore/bib/book
where $book/@date > 2000
return $book/author
)
```

```
<xs:element name="bib">
<xs:complexType>
<xs:sequence>
<xs:element name="book"
      minOccurs="1" maxOccurs="unbounded">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="author" type="authorType"
            minOccurs="2" maxOccurs="unbounded"/>
      <xs:element name="title" type="titleType"/>
      <xs:element name="rate" type="rateType"
            minOccurs="2" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="year" type="yearType"
          use="required"/>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 4: XML Schema example I: Tags

```
<xs:simpleType name="yearType">
  <xs:restriction base="xs:integer">
    <xs:enumeration value="1995"/>
    <xs:enumeration value="2005"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="rateType">
  <xs:restriction base="xs:integer">
    <xs:enumeration value="1"/>
    <xs:enumeration value="3"/>
    <xs:enumeration value="5"/>
    <xs:enumeration value="10"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="titleType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="UML"/>
    <xs:enumeration value="XML"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="authorType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Buneman"/>
    <xs:enumeration value="Abiteboul"/>
  </xs:restriction>
</xs:simpleType>
```

Figure 5: XML Schema example I: Values

```
return <names> {$t/title/text()} </names>
```

Here the programmer makes two mistakes. The first is that "date" is not an attribute of "books". Rather than the attribute is called "year". After, the path "$t/title/text()" is wrong because "$t" is bounded to "$book/author" and "$book" is bounded to "$bookstore/bib/book". Thus the full path is "$bookstore/bib/book/author/title/text()", which is a wrong path. The path validator answers with the following message:

```
Schema Validation Error::Wrong path:date
Schema Validation Error::Wrong path:title
```

Example 2. Now, we show an example of property testing. Let us suppose the following query with respect to the XML Schema of Figure 4, in which values have been added (see Figure 5).

```
<bib>
  {
    for $b in $bib//books
    where $b[author="Buneman"]
    return
        <book>
            { $b/title }
            { for $a in $b/author[position()<=2]
                return $a}
            {if (count($b/author) > 2)
                 then <et-al/>
                 else ()}
        </book>
    }
</bib>
```

And let us suppose to test the following output property:

```
every $b in $bib//book
 satisfies $b/author="Buneman"
```

when the following input property is required:

```
every $b in $bib//book
 satisfies count($b/author)<=2
```

The output property requires that one of the authors of the output document is "Buneman", when the number of authors is smaller than two (according to the input property). The tester reports "Ok". It seems that the program is correct and passes the test. However, the program is wrong due to the path "$bib//books". Here, the answer is not empty because the program always returns at least the label "bib". Also, the output property is trivially true: "bib" is empty and thus all of the books have as author "Buneman". This is the reason why path validation is required. We can also test the same output property, in the opposite case of the previous input property (i.e., the number of authors is greater than two), with the paths corrected:

```
every $b in $bib//book
 satisfies count($b/author)>2
```

In this case, the tester reports:

```
Output Property Falsifiable after 512 tests.
Counterexamples:
<bib>
  <book year="1995">
    <author>Abiteboul</author>
    <author>Abiteboul</author>
    <author>Buneman</author>
    <title>UML</title>
    <rate>1</rate>
    <rate>1</rate>
  </book>
</bib>
```

Because in the case of "Buneman" occurs in the third position, his name is replaced by "et-al", and thus the output property is false. An input-output property which can be tested is the following, specifying that in the output document (represented by "$bibo") each book having "et-al" has more than two authors in the input document (represented by "$bibi"):

```
 every $b in $bibo//book
 satisfies (if (count($b/et-al)=1) then
 some $c in $bibi//book satisfies ($c/title=$b/title
 and count($c/author)>2) else true())
```

The tester answers in this case "Ok", while in the case of the following input-output property, specifying that each input book title is an output book title:

```
 every $b in $bibi//book
 satisfies some $c in $bibo//book satisfies
    $b/title=$c/title
```

the tester finds counterexamples, because the query filters Buneman's books.

```
Input-Output Property Falsifiable after 256 tests.
Counterexamples:
<bib>
  <book year="1995">
    <author>Abiteboul</author>
    <author>Abiteboul</author>
    <title>UML</title>
    <rate>1</rate>
    <rate>1</rate>
  </book>
</bib>
<bib>
```

Example 3: Now, we show that the testing of input and input-output properties can fail when either the schema has not been correctly selected or input and input-output have been not correctly selected. In such cases the tester returns "Unable to test the property". In Example 2, XML Schema has to be conveniently selected. When the input property to be satisfied is, for instance:

```
every $b in $bib//book
 satisfies count($b/author)<=2
```

and the *minOccurs* value of "author" is set to three then the tester answers with "Unable to test the property". Also it happens when *minOccurs* of "author" is set to zero, we require "count($b/author)>2" and the number of iteration steps of the tester is set to zero, one or two (see [1] for more details). The same happens to the output and input-output properties. Let us suppose, for instance, we request as input property:

```
every $b in $bib//book
 satisfies $b/author="Buneman"
```

and "Buneman" has not been included in the XML Schema, or *minOccurs* of "author" is set to zero and the number of iteration steps of the tester is set to zero (see [1] for more details).

5. EVALUATION

We have evaluated our tool with the XQuery use cases available in the W3C page[1]. We have checked properties on the queries of Section 1.1.9.1 of the cited repository (only those queries with a single input parameter). Properties have been selected to be representative to the computation of each query. We have tested input, output and input-output properties. We have analyzed the time required for each query and the number of tests. Benchmarks have been made on a 2.66 GHz Inter Core 2 Duo MAC OS machine, with 4GB of memory. We have used the BaseX XQuery interpreter [8]. The tester is able to answer in a short time ranging from 6 to 763 ms. The number of test cases range from 2 to 1,884. Failed testing takes less time because the tester stops when a counterexample is found. The tester and path validator implementation as well as the examples shown in the paper can be downloaded from http://indalog.ual.es/TEXTUAL.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an extension of our tool for testing XQuery programs. We have shown how the proposed tool is able to automatically test input and input-output properties on XQuery programs. Also we have shown how path validation is useful for improving XQuery testing. The proposed tool makes possible to have a stronger confidence about the correct behavior of the program. As future work, we would like to extend our work as follows. Firstly, there is a natural extension to cover programs with more than one input document. The current implementation of the tool works with any kind of XQuery expression but having a single input document. Secondly, we would like to extend our implementation of path validation with non-abbreviated XPath syntax and other XML Schema statements (types and cardinality). Also the path validator can be improved by showing the location of wrong paths and it could provide recommendations. Finally, we would like to extend our work to the case of white box testing: automatic generation of test cases from the program.

[1]http://www.w3.org/TR/xquery-use-cases/.

7. REFERENCES

[1] J. M. Almendros-Jiménez and A. Becerra-Terón. XQuery Testing from XML Schema Based Random Test Cases. In *Database and Expert Systems Applications*, volume LNCS 9262, pages 268–282. Springer, 2015.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[3] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM (JACM)*, 55(2):8, 2008.

[4] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN notices*, 46(4):53–64, 2011.

[5] S. Cohen. Generating XML structure using examples and constraints. *Proceedings of the VLDB Endowment*, 1(1):490–501, 2008.

[6] P. Genevès and N. Layaïda. A system for the static analysis of XPath. *ACM Transactions on Information Systems (TOIS)*, 24(4):475–502, 2006.

[7] J. Groppe and S. Groppe. Filtering unsatisfiable XPath queries. *Data & Knowledge Engineering*, 64(1):134–169, 2008.

[8] C. Grün. BaseX. The XML Database, 2015. http://basex.org.

[9] J. Hidders. Satisfiability of XPath expressions. In *Database Programming Languages*, pages 21–36. Springer, 2004.

[10] J. Melton and S. Muralidhar. XML Syntax for XQuery 1.0 (XQueryX) (Second Edition), 2010. http://www.w3.org/TR/xqueryx/.

[11] M. Papadakis and K. Sagonas. A PropEr Integration of Types and Function Specifications with Property-Based Testing. In *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*, pages 39–50, New York, NY, Sept. 2011. ACM Press.