

# Generating Test Cases for XML-based Web Component Interactions Using Mutation Analysis\*

Suet Chun Lee  
BUSINEX, Inc.  
7950 Silverton Avenue Suite 206  
San Diego CA 92126 USA  
suetl@businex.com

Jeff Offutt  
Department of Information and Software Engineering  
Software Engineering Research Laboratory  
George Mason University  
Fairfax, VA 22030-4444 USA  
ofut@ise.gmu.edu

*The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01), pages 200–209, Hong Kong, PRC, November 2001.*

## Abstract

*Web software systems are built using heterogeneous software components. They interact by passing messages that exchange data and activity state information. Such heterogeneous message transfers can be structured using the eXtensible Markup Language (XML), which allows a flexible common data exchange. Parsers have been developed to check syntax of component interactions, but there are as yet no techniques for checking the semantic correctness of the interactions. This paper presents a technique for using mutation analysis to test the semantic correctness of XML-based component interactions. In this paper, the web software interactions are specified using an Interaction Specification Model (ISM) that consists of document type definitions, messaging specifications, and a set of constraints. Test cases are XML messages that are passed between the web software components. Classes of interaction-specific mutation operators are introduced and applied to the ISM to generate mutant interactions and test cases.*

## 1. Introduction

As the world wide web matures, web sites are interacting with each other using digitized information that is exchanged through the internet network. This information exchange is especially prevalent among businesses, and includes commercial web sites that interact with customers and business-to-business (B2B) web sites. A recent study from the U.S. National Research Council found that the

current base of science and technology is inadequate for building systems to control critical software infrastructure [22]. This same conclusion was reached by the U.S. President's commission on critical infrastructure protection in the PITAC report [19]. This inadequacy is particularly severe in the novel, high-risk area of web-based software systems. A number of emerging technologies are being used in this segment of the industry, and there is still little known about how to ensure the reliability of web-based software systems. The goal of our current research project is to fundamentally improve the current base of web-based software technology by addressing the problem of ensuring that the technologically diverse, multi-platform software pieces that comprise web applications conform to attributes such as reliability, security, availability, maintainability, efficiency, and interoperability.

Web-based software systems are constructed by integrating a number of diverse components from a variety of sources. Some of these components are built as special-purpose applications, some are pre-existing, off-the-shelf software components built in-house, and some are purchased from third party vendors. Much of the new complexity found with web-based applications is a result of the way the different types of software components are pieced together. There has been a lot of research dedicated to the testing of the individual software components, however there has thus far been little research on the testing of component interactions on heterogeneous web platforms. To assure the quality of such a system, novel techniques are needed to integrate and evaluate the resulting connections of the various components.

This paper focuses on one aspect of this large problem, that of validating the reliability of data interactions among web-based software system components. One of the key technologies that is being used to transmit data among heterogeneous software components on the web is the eXten-

\*This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111.

sible Markup Language (XML) [3, 1]. Briefly, XML is a markup language for describing and formatting data. XML provides an abstract, uniform way to represent many different kinds of data by using a formal language. The data is stored as simple strings that are surrounded by “meta-data”, or *tags* that describe the semantics of the data, making it a method for abstracting many types of data structures. This makes XML ideal for transmitting many diverse types of data among heterogeneous software components, thus offering a simple elegant solution to a problem that has plagued software developers for decades. Thus, part of our concept, that of mutating the data that is transmitted among software components, could be used in non-web applications and even applications that do not use XML. But the formalized structure of XML files allows the concepts to be applied uniformly in an abstract way.

Current web sites are composed of a number of interacting software and hardware components. These components are often arranged in a *multi-tier* architecture, where each component communicates only with components in the two adjacent tiers. Figure 1 illustrates a common model of web site software. Instead of a simple client-server model, the configuration has expanded to a multi-tier model. A client is still a browser used by a person to visit web sites, which are hosted and delivered by web servers. But to increase such factors as security, reliability, availability, and scalability, as well as functionality, most of the software is on a separate computer, the *application server*. Indeed, on large web sites, the application server is actually a **collection** of application servers that operate in parallel. The application servers typically interact with one or more *database servers*, often running a commercial database. One of the primary vehicles for transmitting data between components, both within and across tiers, is now XML [11].

Although XML provides a very convenient way to transfer data, the number and diversity of software components in modern web site software systems introduce a large number of chances for errors. One difficult problem that has yet to be solved is that of ensuring that this diverse set of components interacts properly, particularly with respect to whether the data is transmitted and handled correctly. Several XML parsers are available that can check that XML messages are *well formed* (that is, they obey XML syntax requirements) and *valid* (that is, they follow grammar rules that are defined for that particular XML document) [3, 1]. This makes the **syntax** checking of the web component interactions trivial. However, there has been little research on ensuring the **semantic** correctness of the XML-based web component interactions. This paper presents a technique for checking semantic correctness of web component interactions. Our initial focus is on validating the XML data interactions. We have not, as yet, addressed issues of concurrency.

## 1.1. Organization of the Paper

This paper is organized as follows: Section 2 presents the background knowledge and assumptions. Section 3 presents a model for specifying web component interactions using XML Data Type Definition and an accompanying set of formal constraints (ISM). The ISM presented in this section is used to model the Interaction Mutation Analysis in Section 4. An initial set of IMO classes is then defined, which is the class of mutation operators useful for generating mutant web component interactions. Section 5 illustrates the testing approach with a complete example.

## 2. Background

The research presented in this paper is based on a number of existing research and technological solutions. This section gives a brief overview of XML, which is used as the primary source for the definitions of our test criteria. This paper presents a technique for testing software that uses XML by adapting mutation testing.

### 2.1. Web Components and the eXtensible Markup Language

A *web component* is defined as any software process or combination of processes that execute in the world wide web environment. This definition includes a large variety of types of software, including Java server pages, Java servlets, JavaScripts, Active server pages, Java beans and non-bean classes, databases, and others. Although there is often a strong association between software components and hardware components (as emphasized by Figure 1), this paper restricts attention only to software aspects. Whether two communicating software components are executing on the same or different hardware components is not considered. The current research is restricted to web components that interact by using XML messages, and primarily messages that are transmitted with the HTTP protocol [21].

The eXtensible Markup Language (XML) is designed to be a universal format for structured documents and data on the web. XML is specified by the World Wide Web Consortium (W3C) and is intended to enable data to be exchanged among heterogeneous software and hardware platforms conveniently and easily [3, 1]. XML documents are plain text files, are easy to generate, and can be read by a human or computer. XML uses *tags*, which are textual descriptions of data enclosed in angle brackets (`<` and `>`), to delimit and structure pieces of data. Although the tags are intended to provide semantic descriptions of the data, the interpretation of the data is up to the application that reads it. XML documents follow simple grammars that can be spec-

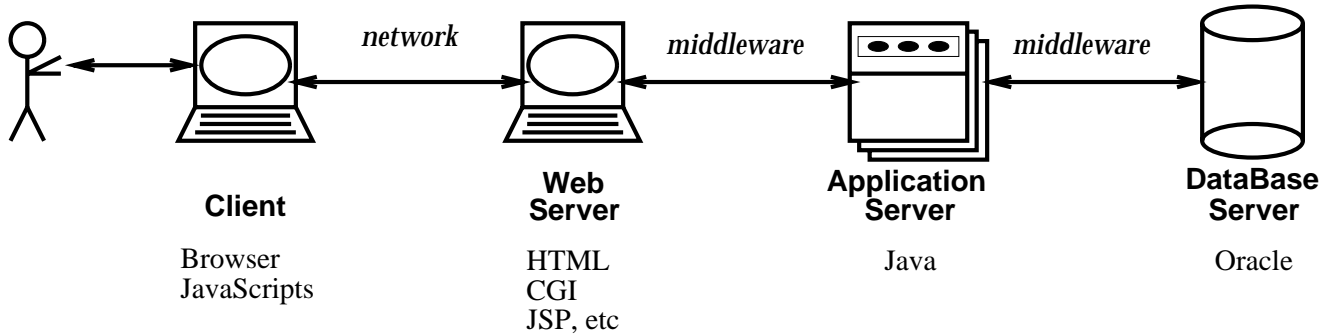


Figure 1. Modern Web Sites

ified using a Document Type Definition (DTD) [1, 3]. An example XML document for computer accounts is:

**Example 1** : Simple XML for Computer Accounts

```
<?XML version = "1.0">
<AUTHORIZED_USERS>
  <AUTHORIZED_USER>
    <USER_ID>jenny</USER_ID>
    <PASSWORD>jen</PASSWORD>
  </AUTHORIZED_USER>
</AUTHORIZED_USERS>
```

The tags are case sensitive, but free form. It is required that each tag have an ending tag (for example <USER\_ID> and </USER\_ID>), and the nesting of tags must not overlap; an XML document that follows the syntactic rules is said to be *well formed*. If a DTD exists, the XML must also follow the grammar rules that it specifies to be *valid*. An example DTD for the above XML is:

**Example 2** : Simple DTD for Computer Accounts

```
<!ELEMENT AUTHORIZED_USERS (AUTHORIZED_USER)*>
<!ELEMENT AUTHORIZED_USER (USER_ID, PASSWORD)>

<!ELEMENT USER_ID (#PCDATA)>
<!ELEMENT PASSWORD (#PCDATA)>
```

The parentheses indicate nesting, the '\*' symbol indicates 0 or more occurrences, and #PCDATA indicates the tags contain character data. This DTD states that an instance of AUTHORIZED\_USERS element in an XML document can contain 0 or more instances of AUTHORIZED\_USER elements, and each instance of an AUTHORIZED\_USER element contains an instance of a USER\_ID element and PASSWORD. If an XML document follows this simple regular grammar, it is said to be *valid*.

XML is used to send messages between independent software components on the web. This is called *peer-to-peer web component communication*, because the communication is between components of equal status and neither is controlling the other (that is, this is not necessarily a traditional client-server interaction). There are currently a num-

ber of industry groups (such as the W3C) that are developing protocols to *envelope* XML messages to enhance robustness, simplicity, reusability and interoperability. As these protocols are still evolving, this paper does not address the message envelope format and transmission protocol specific information [2].

**2.2. Testing and Mutation Analysis**

An XML document can be checked for validity and well-formedness, but these are syntactic issues, and any faults found are syntactic faults. This research targets what we call *semantic faults*, which cause programs to produce incorrect outputs. A test case is an individual XML document, which is created from a DTD to evaluate the functional behavior of component interaction. Test cases are generated by a novel application of mutation analysis.

Mutation is a fault-based testing technique introduced by DeMillo et al. [6, 7]. Mutation testing is based on the assumption that a program will be well tested if all simple faults are detected and removed. The coupling effect [6, 17] states that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults.

Mutation analysis induces faults into software by creating many versions of the software, each containing one fault. The faults are defined by *mutation operators* and each change or *mutation* created by a mutation operator is encoded in a *mutant program*. Test cases are used to execute these faulty programs with the goal of distinguishing the faulty programs from the original program. A mutant is said to be *killed* if the output of the mutant differs from the output of the original program. A test case that kills a mutant is considered to be effective at finding faults in the program, and the mutants it kills are not executed against later test cases. Mutants either represent likely faults, a mistake the programmer could have made, or they explicitly require a typical testing heuristic to be satisfied, such as execute every branch or cause all expressions to become zero.

### 3. Interaction Specification Model

Software specifications for web component interactions are usually written informally in human readable natural languages [24]. To automatically verify whether components conform to their specifications, we need a formal model of the software specification. We introduce an Interaction Specification Model (ISM) that uses a set of formally defined DTDs, messaging interfaces, and constraints.

Fan and Siméon have extended the Document Type Definitions of XML Documents with a set of integrity constraints to preserve the semantics of data originating from relational or object databases [10], called a *DTD Structure*. We extend their DTD structure and integrity constraints to create the interaction specification model.

Following Fan and Siméon, we assume the existence of a set  $\mathbf{E}$  of element names, a set  $\mathbf{A}$  of attribute names, and a set  $\mathbf{S}$  of attribute string values. An XML document can be structured in a tree (the *data tree*), with vertices  $\mathbf{V}$  representing element names, attribute names, and at the leaf level, string values. The element names  $E \subset \mathbf{E}$  and attribute names  $A \subset \mathbf{A}$  are the names from a specific DTD.

Again following Fan and Siméon, for any  $\tau \in E$ ,  $ext(\tau)$  is the set of nodes in the graph with the label  $\tau$ . For any  $x \in V$  and  $l \in A$ , the value of the attribute  $l$  of  $x$  is  $att(x, l)$ , or  $x.l$ . All such values are saved in the set  $ext(\tau).l = \{x.l \mid x \in ext(\tau)\}$ . Furthermore, if  $X$  is the sequence of attributes  $(l_1, \dots, l_n)$ , then the specific sequence of attributes of  $x$  is  $x[X] = (x.l_1, \dots, x.l_n)$ . We adapt Fan and Siméon's DTD structure for the purposes of testing by eliminating the elements that are specific for database and not needed for our purposes.

**Definition 1** A DTD structure is denoted by  $S = (E, P, R)$ , where:

- $E$  is a finite set of *element* types in  $\mathbf{E}$ , whose elements are labeled by  $\tau$ .
- $P$  is a function from element types to *element type definitions*,  $P(\tau) = \alpha$ , where  $\alpha$  is a regular expression defined as follows:

$$\alpha ::= S \mid e \mid \epsilon \mid \alpha + \alpha \mid \alpha, \alpha \mid \alpha^*$$

$S$  is the type of atomic values given above,  $e \in E$ ,  $\epsilon$  denotes the empty element, '+' stands for union, ',' for concatenation, and '\*' for the Kleene closure.

- $R$  is a partial function from  $E \times \mathbf{A}$  to attribute type definitions:  $R(\tau, l) = \beta$ , where  $\tau$  is an element name in  $E$ ,  $l$  is an attribute in  $\mathbf{A}$ , and  $\beta$  is either  $S$  or  $S^*$ . We use  $Att(\tau)$  to denote the set of attributes of  $\tau$ , i.e.,  $Att(\tau) = \{l \in \mathbf{A} \mid R(\tau, l) \text{ is defined}\}$ . An attribute  $l$  is called a *set-valued attribute* of  $\tau$  if  $R(\tau, l) = S^*$ , and single-valued if  $R(\tau, l) = S$ .

This paper introduces two new constraints for XML-based web component interactions, *memberOf* and *lenOf*. These are defined in a language  $L_M$  for constraining web component interactions. Additional constraints for specifying web component interactions will be added to  $L_M$  in the future. *memberOf* is intuitively defined as follows. Given two sequences of elements, the elements in one ( $X$  in Definition 2) forms a subset of the elements in the other ( $Y$  in Definition 2). For example, the instances of element `USER_ID` and `PASSWORD` are in the `AUTHORIZED_USER` element of Example 1. They may also be part of another element, for example `SIGNON_REQUEST` of Example 3. More formally,

**Definition 2** For a DTD structure  $S = (E, P, R)$ , a *memberOf* constraint of  $L_M$  has the form:

$$\tau [X] \subseteq \tau' [Y]$$

where:

- $\tau, \tau' \in E$ .
- *elem* is as defined in Fan and Siméon; a function that maps vertices of the data tree to their labels and their children.
- $X$  is a sequence of elements in  $elem(\tau)$ .
- $Y$  is a sequence of elements in  $elem(\tau')$ .

A *memberOf* constraint,  $\tau [X] \subseteq \tau' [Y]$ , is  $\forall x \in ext(\tau) \exists y \in ext(\tau')^1. x[X] = y[Y]$ .

*LenOf* is intuitively defined as the number of characters in the string representation of an element. More formally,

**Definition 3** For a DTD structure  $S = (E, P, R)$ , a *lenOf* constraint of  $L_M$  has the form:

$$lenOf(\tau[X]) = K$$

where:

- $X$  is a sequence of elements in  $elem(\tau)$ .
- $K$  is an integer constant.
- $\tau \in E$ .
- $ext(\tau)$  is the set of nodes labeled  $\tau$  in the tree.
- $|S|$  is the number of characters in the string  $S$ .

For all  $x \in ext(\tau)$ ,  $|x[X]| = K$ .

For each node in the data tree, the function *elem* provides its label (an element name from the DTD) and its list of children (either string values or nodes in the tree). Extending the DTD for computer accounts in Example 2 to allow for password authentication (`AUTHENTICATION` is an *attribute* of `SIGNON_RESPONSE`), and adding a request yields:

**Example 3** : *DTD and XML files for Computer Accounts*

<sup>1</sup>This is very similar to Fan and Siméon's foreign key constraint, except their  $X$  is a sequence of attributes, whereas our  $X$  is a sequence of elements.

```

<!-- DTD for Computer Accounts -->
<!ELEMENT USER_ID (#PCDATA)>
<!ELEMENT PASSWORD (#PCDATA)>

<!-- Collection of authorized users -->
<!ELEMENT AUTHORIZED_USERS (AUTHORIZED_USER)*>
<!ELEMENT AUTHORIZED_USER (USER_ID, PASSWORD)>

<!-- XML request sent, SignOn A -->
<!ELEMENT SIGNON_REQUEST (USER_ID, PASSWORD)>

<!-- XML response sent, Authenticate B -->
<!ELEMENT SIGNON_RESPONSE (USER_ID)>
<!ATTLIST SIGNON_RESPONSE AUTHENTICATION
    (ALLOW | DENY) #REQUIRED>

<-- XML for Computer Accounts -->
<?xml version=1.0?>
<AUTHORIZED_USERS>
  <AUTHORIZED_USER>
    <USER_ID> Jenny </USER_ID>
    <PASSWORD> jen </PASSWORD>
  </AUTHORIZED_USER>
  <AUTHORIZED_USER>
    <USER_ID> Jeff </USER_ID>
    <PASSWORD> ajo </PASSWORD>
  </AUTHORIZED_USER>
</AUTHORIZED_USERS>

<-- XML for Authorization Request -->
<?xml version=1.0?>
<SIGNON_REQUEST>
  <USER_ID> Jenny </USER_ID>
  <PASSWORD> jen </PASSWORD>
</SIGNON_REQUEST>

```

SIGNON\_REQUEST and SIGNON\_RESPONSE are two complementary XML elements; SIGNON\_RESPONSE allows or denies the request. SIGNON\_RESPONSE has a required attribute, AUTHENTICATION, which has one of the values ALLOW or DENY. The elements for the DTD are:

$$E = \{ \text{USER\_ID}, \text{PASSWORD}, \text{SIGNON\_REQUEST}, \text{SIGNON\_RESPONSE}, \text{AUTHORIZED\_USER}, \text{AUTHORIZED\_USERS} \}$$

For the *memberOf* constraint:

- $\tau = \text{USER\_ID}$  (in the XML document tree with the parent node named SIGNON\_REQUEST illustrated in Example 3)
- $\tau' = \text{USER\_ID}$  (in the XML document tree with the parent node named AUTHORIZED\_USER illustrated in Example 3)
- $\text{elem}(\tau) = \text{Jenny}$  (in the XML document tree illustrated in Example 3)
- $\text{elem}(\tau') = \text{Jeff}$  (in the XML document tree illustrated in Example 3)
- $X = (\text{Jenny})$
- $Y = (\text{Jeff})$
- $\text{ext}(\tau) = \{ \text{USER\_ID} \}$ , where USER\_ID is from the tree
- $\text{ext}(\tau') = \{ \text{USER\_ID} \}$ , where USER\_ID is from the tree

- $x = \text{USER\_ID}$
- $y = \text{USER\_ID}$
- $x[X] = \text{Jenny}$
- $y[Y] = \text{Jeff}$

$x[X]$  is **not** equal to  $y[Y]$ , therefore the *memberOf* constraint is **not** satisfied.

For the *lenOf* constraint:

$$\begin{aligned} \tau &= \text{PASSWORD} \\ X &= (\text{jen}) \\ \text{ext}(\tau) &= \{ \text{PASSWORD} \} \\ x &= \text{PASSWORD} \\ x[X] &= \text{jen} \\ |x[X]| &= 3 \end{aligned}$$

If  $K$  equals 8, then  $|x[X]|$  is **not** equal to  $K$ , so the *lenOf* constraint is **not** satisfied.

These definitions allow the interaction specification model to be defined. Additional constraints for specifying web component interactions will be added to  $L_M$  in the future. The model assumes two software components that communicate via XML, the *client* and the *server*, and the XML is defined by a DTD. The communication from the client to the server is defined by the *request*, and the communication from the server to the client is defined by the *response*. The request and the response are defined in the DTD as in Definition 1.

**Definition 4** The Interaction Specification Model (ISM) is defined as:

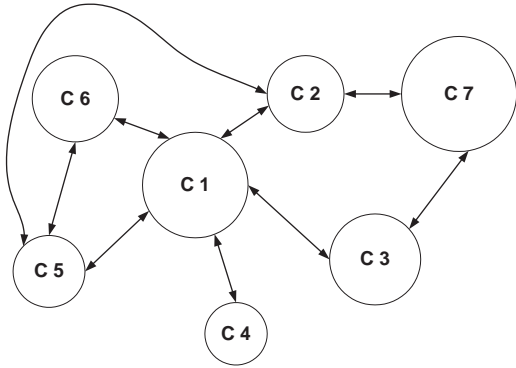
$$ISM = (S, M, \Sigma)$$

where:

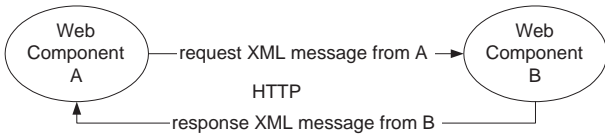
- $S$  is a DTD structure.
- $M$  is a pair of messages,  $M = (\text{request}, \text{response})$  where *request* and *response* are both defined in the DTD structure  $S$ . *Request* and *response* define the grammar for XML messages for the component interaction.
- $\Sigma$  is a set of basic XML constraints expressed in the XML-based web component interaction constraint language  $L_M$ .

## 4. Interaction Mutation Analysis

Web software components interact by passing messages that exchange data and activity state information. Web component interact with other components asynchronously and synchronously. Figure 2 illustrates a system that has seven interacting web components, with arrows to indicate possible messages. However, these *multilateral* interactions are an aggregation of multiple peer-to-peer interactions. Hence, this research paper focuses on evaluating the correctness of peer-to-peer interactions (that is, between individual pairs of components).



**Figure 2. Multilateral Web Component Interaction**



**Figure 3. Peer-to-Peer Web Component Interaction Model**

In Figure 3, **A** and **B** agree to communicate through message passing using XML. **A** and **B** discover information about each other, access data from each other, and interact with each other through Universal Resource Locators (URLs). **A** and **B** can reside within the same web server or on different web servers [2].

We present a novel application of mutation analysis [6, 7] as the basis for our testing method. Mutation analysis has usually been used at the unit testing level [7, 18]. More recently, mutation analysis has been applied to formal specifications [8, 9, 16], software interfaces [4, 5, 13, 14], object-oriented software [15, 25, 26], and protocol testing [20]. Except for the formal specifications, the previous mutation analysis applications have involved modifying the source code. Specification-based mutation analysis techniques have mutated the specifications and helped the tester create test data that would demonstrate that a resulting implementation is incorrect.

The closest work to ours is that of interface mutation [5, 13, 14]. Interface mutation is an integration testing technique that attempts to verify that data is transmitted correctly when passed as parameters, global variables, and return values [5]. Delamaro, Maldonado, and Mathur defined 33 mutation operators that make source level changes to variables, expressions, and statements involved in transmitting data between components. Interaction mutation differs from interface mutation in that we are mutating a well-defined semantic model (the ISM), and using those mutants as models to create instances of mutants in the form of XML messages. The results are new versions of actual **data** that are passed instead of the new versions of the **code**

that passes the data. We are creating the instances by making changes to the data *in transit*, independently of the software components (whose source code is not assumed to be available)<sup>2</sup>.

The source for the web components is often not disclosed due to vendor or internal confidentiality and security restrictions [25]. Thus, source-level mutation is not always feasible. However, the components need to maintain transactional integrity across components, thus the semantics of the web data interactions must be well defined [2]. To use this information, we extend mutation testing to *Interaction Mutation*. Interaction Mutation analysis, abbreviated IM, is performed by mutating the semantics of the component data interactions, specifically by creating mutants of XML messages.

This process is illustrated in Figure 4. On the top line of Figure 4, the two web components C1 and C2 interact through an interaction  $I$  that is specified in an ISM and implemented as XML messages.  $T$  is a set of test cases represented as XML messages that are generated externally. We assume that the interaction  $I$  produces the expected semantic behavior on each member of  $T$ , and that any interaction that produces a different response is faulty.

On the bottom line of Figure 4,  $\Sigma$  is used to create mutation operators by replacing the constraints in  $\Sigma$  with IMOs, yielding  $ISM'$ . The Interaction Mutation System (IMS) generates mutant interactions in the form of mutated versions of  $T$ . When C1 sends a test case  $t$  to C2, the IMS captures  $t$ , applies IMO of  $\Sigma$  of  $ISM'$  to generate a mutant interaction  $I'$ , which is in the form of an XML message ( $t'$ ). The IMS then sends  $t'$  to C2, which processes the message and generates a response. The response is captured by the IMS, and if it differs from the response produced by  $t$ , the mutant is killed. The same process is repeated for each  $ISM'$  on each member of  $T$ , stopping either when the mutant is killed or when  $T$  is exhausted.

#### 4.1. Interaction Mutation Operators

In traditional mutation analysis, mutant programs are generated by applying mutation operators to source code. Each mutation operator can be viewed as a one-to-many mapping rule that takes a program and produces a sequence of alternative programs. For Interaction Mutation analysis, we generate a set of mutant XML messages by applying web component Interaction Mutation Operator (IMO) rules.

Traditional programming language constructs are defined by a grammar. Mutation operators are defined in terms of the language constructs in the grammar. For example, relational operators are mutated by mutation operators such

<sup>2</sup>It was suggested that interface mutation could be applied without changes to the source, perhaps using CORBA, but we have not found a published reference to the idea.

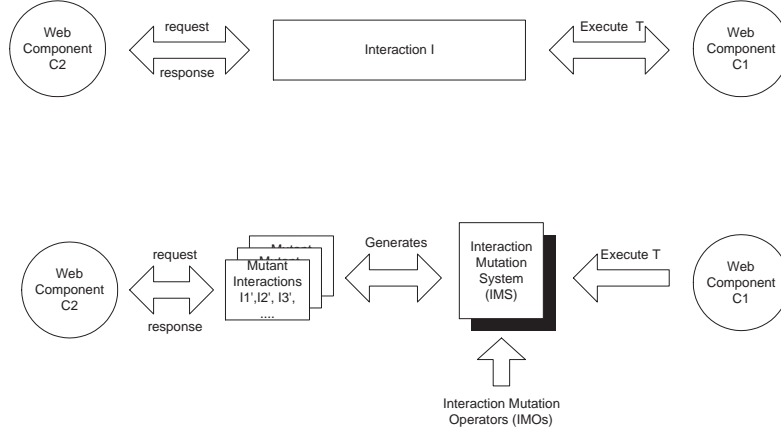


Figure 4. Interaction Mutation Analysis Model

as *ror* (relational operator replacement), which causes each relational operator to be replaced by each other relational operator.

XML is at a higher level of abstraction than programming languages. XML allows users to create new tags for the current language that are useful for the application domain [1, 3]. For example, an XML file relating to an application in the field of mathematics will be constructed using a different data type definition than that of an XML file used for applications in the field of music. Thus, the grammars will be different, and we cannot define data type definition specific mutation operators that work across different XML application domains.

Instead, we define **generic classes** of mutation operators that make use of the  $\Sigma$  in the ISM. These mutation operator classes extend across applications that are based on different data type definitions. The generic classes of operators are then instantiated to form specific operators for specific data type definitions.

To demonstrate the feasibility of our approach, we have developed two initial mutation operator classes. The *memberOf* constraint is used to define the *interaction mutation operator* class *NOT memberOf*, and the *lenOf* constraint to define the class *NOT lenOf*. These classes can be thought of as generic producers of mutant operators. DTDs are supplied as inputs, and mutant operators are produced that are specialized for that DTD. The IMS looks at the DTD, finds XML elements on which the constraint applies, and generates specific mutant operators for that constraint and element.

## 4.2. Interaction Mutation Test Process

Our test process for performing Interaction Mutation analysis is relatively straightforward, as illustrated in Figure 5. A peer-to-peer web component interaction  $I$  is de-

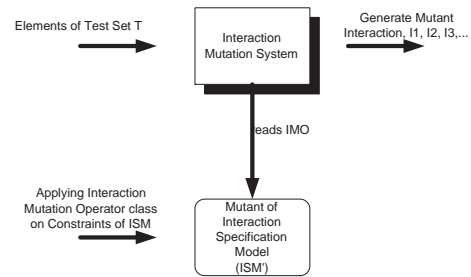
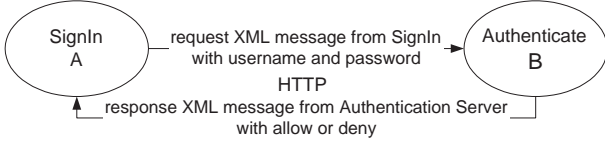


Figure 5. Interaction Mutation Test Process

finied by an ISM, and a set of test cases  $T$  have been generated. No assumptions are made about how the tests were created. We assume that the interaction  $I$  produces the expected semantic behavior on each member of  $T$ , and interactions that produce a different response have revealed a fault. We then generate a number of *mutant interactions*, each of which differs from  $I$  in a small way. We execute each mutant interaction on each member of  $T$ , stopping either when an element of  $T$  is found on which the corresponding mutant interaction produces different responses (it fails), or until  $T$  is exhausted. If the mutant interaction fails on a test, we say the mutant has *died*, otherwise the mutant *lives*. If a large number of mutants live after executing the test data, then either the interaction  $I$  is incorrect or the tests are insufficient.

Interaction Mutation analysis is an iterative process. First an initial set of test cases is generated based on the interaction pair  $M$  and the XML constraints  $\Sigma$ . Then the interaction mutation operators are applied, and the tests are evaluated using the Interaction Mutation System. The tester then examines the remaining live mutant interactions and creates new test cases to try to *kill* them. This process is repeated until all mutant interactions are dead, the tester is satisfied that the remaining mutant interactions cannot be killed (that is, they are different from the original interaction but still correct, that is, *equivalent*), or the tester is out



**Figure 6. Authentication Interaction Model**

of time or budget. A number of empirical studies have supported what Geist [12] called the fundamental premise for traditional mutation: If a fault exists in the program, it is likely that there exists a mutant that can only be killed by a test that also finds the fault. We hope to establish a similar assertion for Interaction Mutation: If a fault exists in the peer-to-peer interaction between two web components, there exists a mutant interaction that can only be killed by a test that also finds the interaction fault.

## 5. Example: Authentication

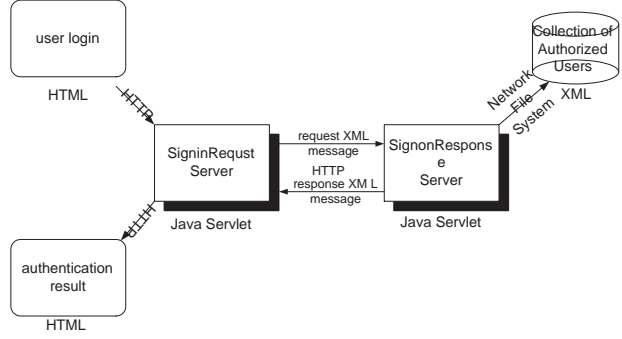
This section presents a fully worked out example of applying IM, using the DTD and XML in Example 3. In our example, we have a web component **A** that offers certain web services that require users to login. Authentication for web component **A** is provided by web component **B**, as illustrated in Figure 6. **A** sends an XML message requesting **B** to verify and authenticate a user who wants to use the services offered. On receipt, **B** does the verification and responds to **A** with a response XML message that denies or allows the user access.

We model the DTD in the ISM as follows. Remember that  $E$  is the set of elements in the DTD, and  $P$  maps element types to element type definitions. Thus, `SIGNON_REQUEST` maps to the two elements `USER_ID` and `PASSWORD`. The *memberOf* constraint, the first element in  $\Sigma$ , says that the values for `SIGNON_REQUEST` must be the same as the values for the `AUTHORIZED_USER`.

```

S = (E, P, R)
E = {USER_ID, PASSWORD, SIGNON_REQUEST,
     SIGNON_RESPONSE, AUTHORIZED_USER,
     AUTHORIZED_USERS}
P(SIGNON_REQUEST) = (USER_ID, PASSWORD)
P(SIGNON_RESPONSE) = (USER_ID)
P(AUTHORIZED_USER) = (USER_ID, PASSWORD)
P(AUTHORIZED_USERS) = (AUTHORIZED_USER)
R(SIGNON_RESPONSE, AUTHENTICATION)=S
M = (request, response)
request = {P (SIGNON_REQUEST)}
response = {P (SIGNON_RESPONSE)}
 $\Sigma = \{SIGNON\_REQUEST (USER\_ID, PASSWORD) \subseteq AUTHORIZED\_USER (USER\_ID, PASSWORD), | PASSWORD [String Value S] | = 3 \}$ 

```



**Figure 7. Test Environment Implementation**

To support our study, we implemented the interaction scenario in Figure 6 using Java Servlets, the Java XML API, and the Tomcat Server. The implementation of our test environment is illustrated in Figure 7. For simplicity, component **A** was designed to be a login screen web component without any other additional functionality. The domain of authorized users was defined using a pre-defined XML data file.

### 5.1. The IMO Definition

$\Sigma$  of our ISM defines the *memberOf* and *lenOf* constraints that are applicable to the example. The mutation operator class “*NOT memberOf*” is defined for Interaction Mutation Analysis to validate the interaction between **A** and **B**. The formal definition is:

$$NOT\ memberOf(\Sigma) = \{SIGNON\_REQUEST (USER\_ID, PASSWORD) \not\subseteq AUTHORIZED\_USER (USER\_ID, PASSWORD)\}$$

The mutant operator class “*NOT lenOf*” is defined as:

$$NOT\ lenOf(\Sigma) = \{| PASSWORD [String Value S] | \neq 3 \}$$

### 5.2. Interaction Test Process

Using the collection of authorized users defined in Example 3, the interaction specification is used to generate the following test case as a request. When the message is submitted to the authentication server **B**, the response from **B** is to allow the signon:

```

Request
<?xml version=1.0?>
<SIGNON_REQUEST>
  <USER_ID>Jenny</USER_ID>
  <PASSWORD>jen</PASSWORD>
</SIGNON_REQUEST>

Response
<?xml version=1.0?>
<SIGNON_RESPONSE AUTHENTICATION="ALLOW">

```

```
<USER_ID>Jenny< /USER_ID>
< /SIGNON_RESPONSE>
```

We then apply the *memberOf* IMO to generate an instance of a mutant of interaction **I** between **A** and **B**, and get the following mutant XML request message. When the message is submitted to the pair of components, the response from **B** is to disallow the signon:

```
Request
<?xml version=1.0?>
<SIGNON_REQUEST>
  <USER_ID>Jeff</USER_ID>
  <PASSWORD>jen</PASSWORD>
</SIGNON_REQUEST>
```

```
Response
<?xml version=1.0?>
<SIGNON_RESPONSE AUTHENTICATION="DENY">
  <USER_ID>Jeff</USER_ID>
</SIGNON_RESPONSE>
```

We then apply the *lenOf* IMO to generate another instance of an interaction mutant, getting the following mutant XML request message. Each XML message is an instance of a mutant interaction *I'*. When the message is submitted to the pair of components, the response from **B** is to disallow the signon:

```
Request
<?xml version=1.0?>
<SIGNON_REQUEST>
  <USER_ID>Jenny</USER_ID>
  <PASSWORD>jenXXX</PASSWORD>
</SIGNON_REQUEST>
```

```
Response
<?xml version=1.0?>
<SIGNON_RESPONSE AUTHENTICATION="DENY">
  <USER_ID>Jenny</USER_ID>
</SIGNON_RESPONSE>
```

In summary, this section illustrates how interaction mutation works. First, an ISM is created for the interaction under test. Then the *NOT memberOf* and *NOT lenOf* IMO classes are created. The test process starts as the IMS applies the IMO classes to generate the mutant ISM, that is, *ISM'*. *ISM'* is a formal representation of the mutant interaction *I'*. Test cases that are instances of mutant interactions are instantiated in the form of XML messages. Each XML message is then sent to the receiving web component. The response from the web component is then captured and evaluated.

If one of the many instances of the mutant interaction is killed, then the mutant interaction is considered to be dead. This implies that the test is able to find the semantic fault

that was modeled by the mutant interaction. On the other hand, if the mutant is not killed, that demonstrates a lacking in the tests. This section illustrated the process with two example mutants; in actual use, there could be many more mutants.

## 6. Conclusions and Future Work

This paper has introduced a novel application of mutation testing to the problem of verifying interactions between heterogeneous web components. An interaction specification model (ISM) for specifying web component interactions was introduced. The ISM provides the elements that are necessary for specifying the test criteria. An initial set of interaction mutation operator classes was also presented. IMOs allow us to dynamically generate mutant interactions that fit the form and function of the application domain. Using the ISM and IMO classes, we presented the interaction mutation test process for verifying the correctness of web component interactions. A real benefit of the XML basis of this work is that XML provides a generic, uniform way to represent many different kinds of data. Thus the concepts in our work can be used in a variety of situations, but the formalized structure of XML files allows the concepts to be applied uniformly in an abstract way.

Further research and identification of the patterns of web component interaction specifications is essential for defining additional classes of interaction mutation operators. Over the next several months, we will be defining new classes of operators for use on XML messages. We also plan to switch from the use of DTDs for defining the grammar of XML messages to the new Schema definition language [23]. We plan to apply our technique to composite multilateral interaction scenarios that can further illustrate the interaction mutation analysis and test process. Furthermore, we plan to develop an interaction mutation test environment that is capable of automatically generating mutant interactions using the ISM.

## 7. Acknowledgments

We would like to thank Dick Lipton of Georgia Tech for first mentioning the idea of using mutation to test web software.

## References

- [1] W. #28. Extensible markup language (XML) 1.0 (second edition) – W3C recommendation, October 2000.
- [2] W. #31. XML messages and how it is being used in data exchange – Simple Object Access Protocol (SOAP) 1.1, 2000.

- [3] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C recommendation, February 1998. <http://www.w3.org/TR/REC-xml/>.
- [4] M. Delamaro, J. Maldonado, and A. Vincenzi. Proteum/IM 2.0: An integrated mutation testing environment. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October 2000.
- [5] M. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.
- [6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.
- [7] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991. <http://ise.gmu.edu/faculty/ofut/rsrch/abstracts/cbt.html>.
- [8] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Mutation analysis testing for finite state machines. In *5th IEEE International Symposium on Software Reliability Engineering (ISSRE '93)*, pages 220–229, Monterey, CA, November 1994.
- [9] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, M. E. Delamaro, and E. W. Wong. Mutation analysis applied to validate specifications based on Petri nets. In *Proceedings of the 8th International Conference on Formal Description Techniques (FORTE'95)*, pages 329–337, Quebec, Canada, October 1995.
- [10] W. Fan and J. Siméon. Integrity constraints for XML. In *19th Symposium on Principles of Database Systems (PODS 2000)*, Dallas, TX, May 2000. ACM.
- [11] S. Feldman. Electronic marketplaces. *IEEE Internet Computing*, 2000.
- [12] R. Geist, A. J. Offutt, and F. Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):550–558, May 1992. Special Issue on Fault-Tolerant Computing.
- [13] S. Ghosh. *Testing Component-Based Distributed Applications*. PhD thesis, Purdue University, West Lafayette IN, 2000.
- [14] S. Ghosh and A. Mathur. Interface mutation. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October 2000.
- [15] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the applicability of traditional test adequacy criteria for object-oriented programs. In *In the Proceedings of the ObjectDays 2000*, October 2000.
- [16] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, December 1999.
- [17] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.
- [18] J. Offutt and R. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.
- [19] P. I. T. A. C. R. T. T. President. Information technology research: Investing in our future. Technical report, National Coordination Office Computing, Information, and Communications, February 1999. <http://www.ccic.gov/ac/report/>.
- [20] R. L. Probert and F. Guo. Mutation testing of protocols: Principles and preliminary experimental results. In I. Davidson and D. W. Litwack, editors, *Protocol Test Systems, III*, pages 57–76. Elsevier Science Publishers B. V. (North-Holland), 1991.
- [21] D. Reinshagen. XML messaging, part I – Write a simple XML message broker for custom XML messages. *Java-World*, 2001.
- [22] F. B. Schneider. *Trust in Cyberspace*. National Academy Press, 1999.
- [23] W3C. XML schema part 0: Primer – W3C recommendation, May 2001. <http://www.w3c.org/tr/>.
- [24] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, 1998.
- [25] H. Yoon and B. Choi. Component customization testing technique using fault injection technique and mutation test criteria. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 93–100, San Jose, CA, October 2000.
- [26] H. Yoon, B. J. Choi, and J. O. Jeon. A UML-based test model for component integration test. In *Component Workshop of the Asian-Pacific Software Engineering Conference '99*, pages 63–70, Japan, December 1999.