Formal Testing of Web Content using TTCN-3

Robert L. Probert, Bernard Stepien, Pulei Xiong
University of Ottawa
(bob | bernard | xiong)@site.uottawa.ca

Abstract

Web applications are both ubiquitous and often unreliable. This, of course, leads to user frustration and dissatisfaction, and steadily lowers the quality bar for web application development and deployment. Part of the problem is the lack of useful research in methods and languages for user-directed testing of web content. At the same time, web content testing has not yet been targeted as a core application domain by TTCN experts. In this paper, we have selected some Web Application (WA) testing requirements recognized as useful in industrial test practice, and show how TTCN-3 is a good platform for implementing these testing requirements. Specifically, we show how part of a TTCN testing template should be made usable by the adaptation layer, and how this technique improves the reusability of test suites for WA testing. The key to the success of our approach involves swapping responsibilities between the abstract test cases and the adaptation layer's encoding/decoding strategies.

IndexTerms: Web Applications, Software Testing, Integration, TTCN-3

1. Introduction

Web applications (**WA**s) are probably the most widespread target domain of software development in computer history since almost everyone, regardless of computer proficiency, has access to them as either a user or a developer or both. However, WAs are also the most interactive applications and most everyone is sooner or later confronted with errors. Although there are no official statistics, web applications appear to have the most widespread distribution of error. For example, a main feature of web applications is the use of links to other pages. These links can point to any other page in the world, thus making the identification of the boundaries of a web application very difficult. The testing of web applications is at least as important as any other type of software testing, but the very distributed nature of web applications makes it difficult to test using traditional testing methods, for example, those using oracles to compare an output to a precise expected output [8]. Another complicating factor is that some web applications, such as on line newspaper, have by definition a constantly changing content. However, they are composed from some fixed information, structure or policies that can still be tested. Finally, many web applications are generated dynamically from information contained in a data base or in a cookie. Generated web pages should always be tested to verify some fixed properties such as structure or formatting rules.

The current state of research in web application (WA) testing emphasizes object oriented concepts in both designing and testing of WAs [1, 4]. Experiments using TTCN-3 for WA have been limited to stress testing [7], XML based web sites testing [6] and hyperlink testing. Content testing has not yet been addressed by the TTCN-3 community despite the fact that the subject has been widely studied by the WA community which resulted even in a proposal of a WA dedicated formalism [1,3].

In this paper we report how we have selected a number of WAs testing requirements proposed by the non-TTCN-3 literature in [1,3] and have studied how these testing requirements can be implemented using TTCN-3. We also focused on the separation of concerns between the abstract and adaptation layers. As a result of our experiments on WA testing, we present the need for an additional capability for TTCN-3, namely the ability to make part of a template usable by the adaptation layer. We have explored how to exploit this capability by trading responsibilities between the abstract test cases and the adaptation layer's coding/decoding strategies. Moreover, this allows better reusability of the abstract (TTCN-3) Test Cases.

2. Selected Web Application Content Testing Requirements

In this section, we outline the primary categories of HTML string content categories, list the most common capabilities of XML based test languages, give an example of XML code for an XML-directed approach, and discuss its shortcomings.

Regardless of how a web page is implemented (HTML or XML), the response that we will parse is always some string streams containing the content showing on the page. This code contains primarily four categories of information: free text, formatting information, links to other web sites and forms. Checking responses to test stimuli involves parsing the response string, then using a checking or matching mechanism to decide whether the response was appropriate and correct. Automated testing consists of writing test scripts and test implementation software using a number of tools such as JUnit, or other WA formalisms as mentioned earlier which provide capabilities to send test strings and to parse and check WA responses.
Automated testing consists of test case design and specification, and test implementation. Generally, the test may be implemented in two ways: GUI Automated testing based on record/playback mechanism, and Non-GUI automated testing by writing test scripts in script languages such as TCL/TK, Unix/Linux Shell or Dos Command Shell etc., or building test programs in programming languages such as Java, C etc.
Several authors have proposed testing these categories of information using some formalism for the test specification based on XML [1]. Using XML enables considerable savings in parsing effort by using available XML tools. The most frequent capabilities of such XML-based test management languages are as follows:

- Link checking: checking that all links on a page are active: This might be performed recursively according to a fixed depth to avoid cycles.
- Text fragment checking using regular expressions: checking if specific words or phrases are present on the page, possibly in a given order.
- Protocol checking: checking if the links of a page are absolute or relative or merely homogeneous according to some policy.
- Table checking: checking if a table is present and has the expected number of

rows and colums and pre-defined properties. Again, some of these tables might have been generated dynamically.

- Form checking: checking if a given form is present, if all the expected fields are present and of specified length, and if the send button link is accurate.
- Formatting checking: verifying text fragments against formatting expectations. This differs slightly from simple text checking in that the formatting tags can be located loosely on the page as opposed to a fixed string for text content.
- Mirror sites content checking: mirror sites are supposed to be identical by definition. However, mirror sites are mostly updated in a separate process from the primary site. In the meantime, values could have changed taking mirror sites out of sync with the main site.
- Development and production sites content checking: This is somewhat equivalent to the mirror sites problem but with some differences.
- Invalid HTML Syntax checking: this is particularly important in the case of generated HTML.
- wrong version checking: e.g., of an inappropriate page for web pages that are available for different languages.
- cookie expiration checking for cookie-based page transitions.

While most of the above described tests can be performed as unit tests, in some cases, when the same entered data should appear on several subsequently linked pages, there is a need to perform an integration test. Also, the testing of a WA does not consist of choosing the most appropriate testing feature. In some cases, several testing features have to be applied to the same application to ensure full conformance to requirements. For example, performing a recursive hyperlink check does not guarantee that the links of a web page are the correct ones in the first place. This only verifies that the links are active. However, as is often the case for multilingual web sites, a French page may have links pointing to the English version of the linked page. Thus, once the integrity of a link is verified, a second test has to be made on the same content, but this time using some text value that can positively determine if the appropriate version has been specified in the link (using some text values that are unambiguous such as different languages).

While testing of WAs is often performed manually, many authors have seen the need for automation, and have realized the benefits of defining a formalism to do so [1]. However, none of these authors seem to refer to TTCN-3 as a solution. It is unclear whether this is due to lack of awareness or for some other reason.
Most of these formalization attempts consisted of defining a language to structure and manage test suites with test cases and test steps, but also defined a number of keywords to specify test elements to address specific functionalities of a WA.

Since creating a new language requires the development of new tools, most of these proposals reduced the tool development effort by encoding their test scripts in XML, thus saving themselves the cost of development of a parser.

A typical test specification for one of the above categories might look like the following code proposed by [1]:

```
<testcase name="Content check using predicates">
      <teststep name="Content check step one">
            <request url="http://www.cs.depaul.edu/program"/>
            <response statuscode="200">
                  <and>
                        <or>
                              <match op="contains"
                                    regexp="false"
                                    select="/html/body"
                                    value="Undergraduate
                                                Degree"/>
                              <match op="contains"
                                    regexp="false"
                                    select="/html/body"
                                    value="Bachelor Degree"/>
                        </or>
                        <match op="contains" regexp="true"
                              select="/html/body”
                              value="[M|m]aster [D|d]egree"/>
                  </and>
            </response>
      </teststep>
</testcase>
```

The above example illustrates the use of two keywords for specifying *requests* and corresponding *responses*. In addition, the important test concept of matching mechanism is provided (keyword *match*), including specification of attributes such as the kind of matching (keyword *contains*), the location where the values are to be found (keyword *select*) and the actual values (keyword *value*) which specifies the strings Undergraduate Degree, Bachelor Degree and M(m)aster D(d)egree. There also seems to be some concept of regular expression since the values for "Master degree" are allowed to start with either upper or lower case.

While the above representation is formal, the use of XML makes the test somewhat difficult to read, is useable only for one test at a time and doesn't present an overview of the testing of an entire application. More important is the fact that in-house solutions like the above are non-standard, and thus do not benefit from either readily available tools or research efforts by the scientific community.

3. TTCN-3 Web Content Testing Solutions

In this section, we give TTCN-3 implementations for a subset of the testing features listed above, and indicate TTCN-3 extensions needed for both the abstract test specification level and the backend adaptation level.

We provide TTCN-3 solutions for the following features: hyperlink checking and especially the recursive link checking, text fragments checking, protocol checking, form checking, and mirror web sites and development/production checking. These features all illustrate various aspects of TTCN-3 implementation models.

First of all we stress the differences among three different approaches to testing in general and WAs in particular. Automated testing can be achieved either using a traditional programming language like Java or C++ , an application dedicated formalism like the one described in [1,2] or a general formalism like TTCN-3.
The application dedicated formalisms have the disadvantage that their use is limited to the application they are designed for and, in the case of WAs, they are not extensible. General formalisms provide the benefits of greater flexibility and customizability for testing applications. In particular, TTCN-3 as a general testing language allows the user to define test features using data types for the purpose of encoding requests and response processing.

Request and response contents are mapped to TTCN-3 data types, while commands and test data are mapped to TTCN-3 templates. A TTCN-3 template is more than just a test data structure since it incorporates various matching mechanisms. In fact, since it comes with its own, TTCN-3 does not require the development of a matching mechanism.

A particularly interesting aspect of WA testing is that testers can design test scripts from requirements rather than from implementation details. Thus they can work more naturally at the abstract level. This makes the use of TTCN-3 all the more appropriate. The basic idea is that the test cases specify the important elements that should be in WA content while the adaptation layer consists mainly of an HTML parser or scanner that locates these required data elements among the potentially massive amount of content data that would either be irrelevant for the test purpose or plainly not testable because of its volatile nature.

TTCN was initially created to test telecommunications protocols. Thus, the data types of the abstract model are very close to the real data at least at the structural level. The only difference is really the layout of bits in the concrete data. This allows TTCN-3 to address a wider area of applications, since data types represent the essential pieces of information required to draw conclusions from a test. In the case of WAs, data types may contain only a fraction of the data needed for the purpose of the test or even no real data at all but merely boolean values indicating that the adaptation layer parser has found the specified string inside the page. Thus, there is no need to describe all elements of a web page but only the information within specific selected tags. Also, since WAs handle mostly strings, many applications can be handled with a simple string type.

Since TTCN-3 is a formalism dedicated to testing, it is very natural to use for WA testing. However, since TTCN-3 is more general than an XML-based formalism, the possibilities to address other new classes of WA testing problems are potentially unlimited as opposed to a dedicated formalism that would have designed only for a known set of WA problems at a given point in time. Finally, we show that it is more appropriate to use a more general testing language like TTCN-3 because there are no limits to the application area while a dedicated formalism like [1] would be naturally restricted to a specific set of features. In other words, with TTCN-3 you get most of the advantages of a formalism while enjoying the ability to introduce new features.

The basic TTCN-3 building blocks are test suites, test cases, functions, data types, data templates, send and receive statements, and alternative response structures. Even more important is the capability of specifying concrete test data and the explicit matching mechanism. Consequently, our discussions will concentrate on the mapping of web application test elements to these basic TTCN-3 building blocks.

While test suites and test cases can naturally be mapped to identical semantic units of each of the above mentioned other formalisms, TTCN-3 functions are mapped onto test steps as they were in the former version 2 of TTCN. TTCN-3 send and receive operations are mapped on WA requests and responses. Matching rules are all specified within the data templates.

In TTCN-3 there are different ways for specifying requests depending on how many different requests are issued to the same host or with different protocols. The most trivial way to represent a Web URL request is by using the basic charstring type as follows:

```
template charstring program_page_web_url :=
             "http://www.cs.depaul.edu/program";
```

But the above is not very flexible nor re-usable, thus a more sophisticated web request can be issued using a record type:

```
type record URLRequest {
      charstring Protocol,
      charstring Port,
      charstring host,
      charstring ResourcePath
}
```

The same request as above would be encoded in a structured template as follows:

```
template URLResquest RequestProgram := {
      Protocol := "http://", Port := 80,
      host := "www.cs.depaul.edu", ResourcePath := "program"
}
```

A new request can be defined as a simple modification of a previous request as follows:

```
template URLRequest RequestCourses modifies RequestProgram := {
      ResourcePath := “courses”
};
```

Once the test features have been defined, they can be used for requests and responses using a simple send and receive construct as follows:

```
webPort.send(program_page_web_url);
```

The response corresponding to the XML example above is as follows:

First, a response type is defined:

```
type record content_response_type
{
      charstring status,
      charstring content
}
```

The most obvious form of verifying a response in TTCN-3 consists of matching a response template against the data returned from the adaptation layer as follows:

```
template content_response_type content_response_predicate := {
      status := "HTTP/1.1 OK",
      content := "it was a dark and stormy night"
}

alt {
      [] webPort.receive(content_response_predicate) {
            setverdict(pass)
      }
      [] webPort.receive {
            setverdict(fail)
      }
}
```

In the rest of this paper we show that this simple model is not always adequate for WA testing, and we propose various solutions that use powerful features of TTCN-3 to address some of the more sophisticated WA testing problems. For example, so far WA test models have considered relatively simple request/response behaviors. It is unclear from [1] whether this formalism can be used to specify sequences of request/responses and also more important, alternative responses. There also seems to be a lack of awareness of the concept of alternative responses altogether in the WA test community. TTCN-3 is particularly well suited to such a purpose mostly because this is a main requirement for testing telecommunication protocols, the first and very successful application domain for TTCN.

While most testing models are pictured as pairs of requests and responses, in TTCN-3 responses are sometimes more appropriately represented as alternative behaviors. Also, tests written in traditional programming languages or using the formalisms proposed in [1,3] have the common property of performing all matching inside the back end while in TTCN-3 the matching mechanism occurs in the abstract layer itself after receiving the abstract data from the adaptation layer. This structural difference requires innovative solutions and potential TTCN-3 language extensions as discussed in the following sections.

All of the proposed solutions require encoders at the adaptation layer that can be implemented either entirely from scratch or re-use existing HTML parsers and sometimes also converters to HTML when for example a web page is written in javascript, etc… All of these external tools will have to be integrated in the adapter. However these external tools role are limited in selecting data from an HTML page for the only purpose of populating data elements from the abstract data types. The important thing here is to remember that it is the TTCN-3 abstract layer that performs the matching. This is in contrast with the other formalism in [1,3] where matching is performed at what we call the adaptation layer and thus must be implemented by the user rather than be readily available in the tool.

3.1 TTCN-3 Recursive Hyperlink Checking

The first example found in [1] consists of the following formal specification of recursive hyperlink checks:

```
<request url="http://www.cs.depaul.edu", recursive="true",
                                    recursivedepth="3"/>
<response statuscode="200"/>
```

This can be specified naturally in TTCN-3 using functions. The function *CheckChildLink(...)* consists of first sending the URL Link of a specific web page, then getting the list of links that page contains in the response (TTCN-3 receive statement), then invoking recursively this same function to process each element from that list.

```
The main data type used for this purpose is the hyperlink_response_type
that contains two fields, one for the response status and one that
contains a list of the href links found on a web page defined as
follows:
```

```
type record of charstring HrefListType;

type record hyperlink_response_type
{
      charstring status,
      HrefListType hrefList
}
```

The template has a simple value collection functionality:

```
template hyperlink_response_type hyperlink_response :=
{
      status := "HTTP/1.1 200 OK",
      hrefList := ?
}
```

The structure of the test behavior to implement the recursive checking is as follows:

```
testcase Link_Check_TC() runs on MTCType system SystemType {
      ...
      theOverallVerdict := CheckChildLink(main_page_web_url, 1);
      setverdict(theOverallVerdict);
}

function CheckChildLink(charstring theURLLink, integer theDepth) runs
```

```
                              on MTCType return verdicttype {

    var hyperlink_response_type theHyperlink_response;
    var charstring theBadResponse;
    var verdicttype theFinalVerdict := pass;
    var verdicttype oneVerdict;

    log("testing link: " & theURLLink);

    web_port.send(theURLLink);
    alt
    {
          [] web_port.receive(hyperlink_response) -> value
                              theHyperlink_response
             {
                if(theDepth <= maxDepth)
                {
                      var integer theNewDepth;
                      var HrefListType theResponseHrefList :=
                              theHyperlink_response.hrefList;
                      var integer numOfLinks :=
                              sizeof(theResponseHrefList);
                      var integer i;

                      theNewDepth := theDepth + 1;

                      for(i:=0; i < numOfLinks; i:=i+1)
                      {
                            oneVerdict :=
                            CheckChildLink(theResponseHrefList[i],
                                                    theNewDepth);

                            if(oneVerdict == fail)
                            {
                                  theFinalVerdict := fail;
                            }
                      }
                }
             }
          [] web_port.receive(charstring:?) -> value theBadResponse
             {
                log("the URL: " & theURLLink & " did not work,
                              response: " & theBadResponse);

                theFinalVerdict := fail;
             }
    }

    return theFinalVerdict;
}
```

Here, it is the responsibility of the adaptation layer encoder to scan the input stream coming back from the HTTP request and to assemble an abstract list of href specifications that is then processed by the above TTCN-3 abstract code. For example, if the adapter code is written in Java, the standard Java String class methods can be used.

3.2 TTCN-3 Text Fragments Checking

Text fragments checking can be naturally handled by using the pattern feature of TTCN-3 character strings. The pattern is specified at the template level:

```
      template content_response_type content_response_pred :=
      {
            status := "HTTP/1.1 200 OK",
            content := pattern "*<BODY*[[Undergraduate Degree|Bachelor
Degree]*[M|m]aster
      [D|d]egree|[M|m]aster [D|d]egree*[Undergraduate
                        Degree|Bachelor Degree]]*
                                    </BODY>*"
      }
```

The above template is then simply applied to a receive statement as follows:

```
      web_port.receive(content_response_pred)
```

Here we can observe that the two functionalities of the XML formism from [1] represented by their keywords *contains* and *select* are handled by the single TTCN-3 template *pattern* feature. The adaptation layer encoder's role consists simply in returning a string without doing any processing upon it. The TTCN-3 matching mechanism will handle the evaluation of the regular expression, thus saving the test designer the development of such a feature. However, this one shot approach requires two additional alternative receive statements, one to indicate that the status did not match, regardless whether the content for example, in the case of a non existent web site, would not even exist, and finally an alternative to indicate that the status did match but the content did not. This would also require two different templates, one template where the content field matching rule is set to any value: *content := ?* and another template where the content of the status field is set to any value: *status := ?* respectively.

This is one way to implement the above problem in TTCN-3. Another way would consist in using a Boolean template like *has_found_my_string* and let the adaptation layer do the regular expression matching. The encoder would then merely return the value *true* or *false* via the template. This approach could be used when some features are not available in TTCN-3.

3.3 TTCN-3 Protocol Checking

The most natural TTCN-3 matching mechanism feature for WA testing is the concept of comparing string patterns. However, this powerful TTCN-3 feature cannot be used in all cases. For some applications described in [1], such as inspecting collections of patterns, simple string pattern matching is inappropriate. For example, the testing of protocols used in WA *href* links is better handled by first collecting all occurences of *hrefs,* saving them in a list and returning this list to the abstract layer where it can be inspected using TTCN-3 string handling facilities.

Deleted: c

```
testcase Forall_protocol_check_TC(charstring theURLLink, charstring
                        theProtocol) runs on MTCType system SystemType
{
      var hyperlink_response_type theHyperlink_response;
      var charstring theBadResponse;
      var boolean all_match;

      map(mtc:web_port, system:system_web_port);

      web_port.send(theURLLink);
      alt
      {
            [] web_port.receive(hyperlink_response) -> value
                                    theHyperlink_response
               {
                  var HrefListType theResponseHrefList :=
                                 theHyperlink_response.hrefList;
                  var integer numOfLinks :=
                                    sizeof(theResponseHrefList);
                  var integer i;
                  var integer theLength := sizeof(theProtocol);

                  all_match := true;

                  for(i:=0; i < numOfLinks; i:=i+1)
                  {
                        if(substr(theResponseHrefList[i],0,theLength)
                                                != theProtocol)
                        {
                              all_match := false;
                              log("link: " & theResponseHrefList[i] &
                              " is not of protocol " & theProtocol);

                        }
                  }

                  if(all_match)
                  {
                        setverdict(pass);
                  }
                  else
                  {
                        setverdict(fail);
                  }
               }
            [] web_port.receive(charstring:?) -> value theBadResponse
               {
                  log("the URL: " & theURLLink & " did not work,
                                 response: " & theBadResponse);

                  setverdict(fail);
               }
      }
}
```

This example shows also some limitations of the matching mechanism when working with lists. First of all, in this case the length of the list is not necessarily known. Thus a

combination of list matching with a template for individual list elements as for the next Form Checking example in the next section is not possible. Thus, we had to use a sort of traditional computer languages approach to implement the matching mechanism. Here a potential extension of TTCN-3 would consist in specifying that a list should be composed of elements that match a certain pattern like http://* regardless of its length.
However, in our case, the code for collecting the list of *hrefs* from the adaptation layer encoder can be simply reused. Here, the only difference from recursive hyperlink checking is that a simple string comparison is performed on each element of the list to determine the verdict of the test case rather than attempting to verify whether this link is active.

3.4 TTCN-3 Form Checking

Form checking consists of verifying that a form is present and that all fields conform to name, type, length and value specifications. At the abstract layer we use a relatively simple model which has only a common type for all form types. In reality, form types are different and do not have all the same attributes. Thus form checking could be better represented using TTCN-3 unions. This is left as an excercise to the reader.

The following HTML form definition is used as an example:

```
<HTML>
<BODY>
      <H3>Subscription information</H3>
      s<FORM NAME="subscription_form" METHOD=POST
      ACTION="http://www.someone.com/cgi-bin/processSubscription.pl">
      <PRE>
      First Name:       <INPUT NAME="firstname" TYPE="text" SIZE="20"
                                                   VALUE=""/>
      Last Name:        <INPUT NAME="lastname" TYPE="text" SIZE="35"
                                                   VALUE=""/>
      City:             <INPUT NAME="city" TYPE="text" SIZE="40"
                                             VALUE="Ottawa"/>
      </PRE>

      <INPUT TYPE="checkbox" NAME="send_me_emails">Please send me e-
mails about your events</INPUT>
      <br>
      <INPUT TYPE="checkbox" NAME="wine_and_cheese">Yes, would like to
come to the wine & cheese party</INPUT>
      <br>
      <INPUT TYPE="submit" VALUE="send"/>
      <br>
      <INPUT TYPE="reset" VALUE ="Oooops!_Let_me_try_again!"/>
      </FORM>
</BODY>
</HTML>
```

First we define a data type for the form content.

```
      type record FormResponseType
      {
            charstring status,
            FormType form
      }

      type record FormType {
            charstring name,
            charstring method,
            charstring action,
            InputListType inputList
      }

      type record InputType {
            charstring name,
            charstring type_input,
            integer size_input,
            charstring value_input
      }

      type record of InputType InputListType ;
```

Then we define the template to be matched:

```
template FormType theRegistrationForm_template := {
      name := "subscription_form",
      method := "POST",
      action := "http://www.someone.com/cgi-
                              bin/processSubscription.pl",
      inputList := {
                  {name :="firstname", type_input := "text",
                        size_input := 20, value_input := ""} ,
                  {name :="lastname", type_input := "text",
                        size_input := 35, value_input := ""} ,
                  {name :="city", type_input := "text",
                        size_input := 40, value_input := "Ottawa"},
                  {name :="send_me_emails", type_input := "checkbox",
                        size_input := 0, value_input := ""} ,
                  {name :="wine_and_cheese", type_input := "checkbox",
                        size_input := 0, value_input := ""} ,
                  {name :="", type_input := "submit",
                        size_input := 0, value_input := "send"} ,
                  {name :="", type_input := "reset", size_input := 0,
                        value_input := "Oooops!_Let_me_try_again!"}
            }
}
```

In the behavior description, the single receive statement will simply attempt to match the template to the abstract data that has been assembled in the adaptation layer encoder. In this case, the length of the list of form elements is well known and we can directly use the TTCN-3 matching mechanism between the abstract and the concrete list. The use of the TTCN-3 record type implies that the form elements should appear in a strict order. If this order is not significant for the test, a TTCN-3 set type should be used instead.

```
web_port.send(theURLLink);
web_port.receive(theRegistrationForm_template);
setverdict(pass)
```

This example shows the benefits of the separation of concerns between the abstract test case and the adaptation layer. The abstract test case specifies the essential elements of a test case, i.e. that a number of form elements and their properties should be matched. The adapter layer performs all the necessary parsing of the web page and assembles the abstract list of form elements. This can of course be achieved with external readily available tools that can integrated in the adaptation code.

The above should be supplemented with the usual alternatives for error handling.

3.5 TTCN-3 Mirror Sites Checking

Mirror sites checking appears at first glance to be very simple depending whether the content is completely identical on all mirror sites. Unfortunatly, most of the time there are minute differences, like for example, the name, country, and contact information.

The first case can be handled simply by storing the response of the first site in a simple charstring variable and using this variable as a template for the second site as follows:

```
web_port.send(first_page_url_templ);
web_port.receive(first_page_response_templ) -> value
                                          theFirstPageResponse;
mirror_port.send(mirror_page_url);
alt
{
      [] mirror_port.receive(valid_web_response(
                              theFirstPageResponse.content))
         {
            setverdict(pass)
         }
      [] handle_errors(mirror_page_url)
}
```

The second case requires a separation of the components of a page into separate fields. It is then the responsibility of the adaptation layer encoder to determine what part of the page goes to which field. This requires of course that the layout of information in both pages is identical.

```
type record local_web_response
{
      charstring status,
      charstring location_information,
      charstring rest_of_content
}

template local_web_response first_page_local_response_templ :=
{
      status := "HTTP/1.1 200 OK",
      location_information := "New york Office",
		rest_of_content := ?
	}
```

Using a parametric template:

```
	template local_web_response valid_local_web_response(charstring
theExpectedContent, charstring theExpectedLocalInformation) :=
	{
		status := "HTTP/1.1 200 OK",
		location_information := theExpectedLocalInformation,
		rest_of_content := theExpectedContent
	}
	…
```

and the corresponding TTCN-3 behavior code would be as follows:

```
	web_port.send(first_page_url_templ);
	web_port.receive(first_page_local_response_templ) -> value
						theFirstPageResponse;

	mirror_port.send(mirror_page_url);
	alt
	{
		[] mirror_port.receive(valid_local_web_response(
				"Los Angeles office",
				theFirstPageResponse.rest_of_content))
		   {  setverdict(pass)  }
		[] handle_errors(mirror_page_url)
	}
```

In the mirror web site we have specified the expected location information as "*Los Angeles Office*".
In this example, the adaptation layer's encoder will require some ways to separate the fixed from the variable content. This may not be an easy task with HTML because of the absence of context indications. This is however one good reason to favor XML as a WA development language since with XML there would be some totally unambiguous tag indicating what portion is the local contact information as shown in [6].

3.6 On the Need for Multiple Tests

The testing of web pages can rarely be performed in a single step. Different kinds of tests need to be applied to the same web page or web system to ensure conformance to the requirements. For example, a recursive hyperlink check cannot be performed simultaneously with a text fragmentation check. This would require a very complex encoder/decoder that would defeat the goal of clarity of TTCN-3 test suites. Another example is that even if hyperlinks of a web page are alive, the content of the web pages they are pointing to may not be appropriate. Consequently, for WA testing, it is preferable to consider several independent test cases for the same web page. These various test can be managed appropriately using the TTCN-3 control language construct as follows:

```
control {
      verdictype theHyperlinkVerdict := execute(Link_Check_TC());

      If(theHyperlinkVerdict = pass)
      {
            verdictype  theTextFragmentVedict :=
                                          execute(TextFragment_TC());
            if(theTextFragmentVedict = pass)
            {
                  setverdict(pass)
            }
            else  { setverdict(fail) }
      }
      else  { setverdict(fail) }

}
```

4.0 Need for TTCN-3 Parametric Adaptation Layer Extension

An interesting difference between the XML based formalism of [1] and TTCN-3 is that TTCN-3 can not directly give instructions to the adaptation layer about how to parse the incoming data for example by passing values to be searched for on a web page. Traditionally, TTCN-3 adaptation is bound to a specific data type for which there is only one way to code and decode. In WAs, the problem is more oriented towards the extraction of a specific information among otherwise irrelevant information. For example, in [1], there is the concept of quantified predicates to enable checking whether all hrefs match *http://* protocol. If we would like to perform the same test on another protocol such as *javascript://*, then there would only be some unintuitive ways to do so using four different approaches:

1. use regular expressions for simple cases, one at a time.
2. use differentiated data type names for each protocol and resolve the differences at the encoder level.
3. use of encoding variations (TTCN-3 *with* construct).
4. Use a combination of a send template, adaptation layer variables and receive templates.

What is missing in this case from TTCN-3 is a concept of parametric template where the parameters are used not only to instantiate the values of the template, but also to pass on decoding criteria values to the encoder. This is in essence what the formalism in [1] does.

TTCN-3 has the concept of encoding rules using the *with* construct. But this is not flexible enough since for each new value to be used in the encoder there needs to be a declared encoder/decoder. The fourth method while fully feasible has the drawback to drift away from TTCN-3's formalism which somehow defeats the purpose of using TTCN-3. What we propose here is a fully parametric template where the actual value of a parameter is used by the encoder only.

This could be achieved by creating a new keyword **codecparameter** that associated with a field name would enable the codec to use the value declared in a template of this type and prevent the abstract layer to use that value for matching purposes. The following example illustrates this concept.

```
type record myRecord {
      integer field_1,
      charstring field_2 codecparameter,
      charstring field_3
}

template myRecord myTemplate :=
{
      field_1 := 5, field_2 := "http://", field_3 := "abcd"
}
```

The required changes to the codec structure will be addressed in another paper.

Deficiencies of other formalisms

Some basic concepts missing in the XML based formalism of [1] are the concept of test configuration and the related concept of communication port. **These are core parts of TTCN-3.**

Finally another vital semantic feature in TTCN-3, the control section that allows the execution of test cases conditionally, depending on the verdict of other test cases, is completely absent from the formalism in [1].

In summary, the main quality of a language such as TTCN-3 is to separate concerns about various aspects of a test. Test data is defined separately from the actual test sequence specification. The test structure thus becomes very clear especially when using expressive names for data templates. Overall, TTCN-3 is clearly superior as a specification language for WA testing.

Conclusions

We have explored how to test WA content using TTCN-3 with an emphasis on utilizing the powerful built-in features of the language to satisfy the common testing requirements for WA testing, extending some TTCN capabilities where necessary, and adding a new feature, parametric coder/decoder to improve the re-usability of templates and their corresponding codec. We have found that TTCN-3 provides powerful mechanisms for specifying concisely WA testing without getting lost in implementation details.

We are very enthusiastic about the applicability of TTCN-3 to testing web applications, and believe that this domain will be yet another successful application domain for this powerful, testing-directed language.

We thank Testing Technologies IST GmbH, Berlin for providing us the necessary

tools TTthree and µTTman to carry out this research.

Deleted:

References

[1] Xiaoping Jia and Hongming Liu, Rigorous and Automatic Testing of Web Applications, DePaul university
[2] Robert L. Probert, Pulei Xiong, Bernard Stepien, A Life-cycle E-Commerce Testing with OO-TTCN-3, in FORTE'04 workshops
[3] Sahil Thaker, Robust Testing for Web Applications, in devnewz, http://www.devnewz.com/articles/0226a.html
[4] G. A. Di Lucca, A. Fasolino, F. Faralli, U. De Carlini, Testing Web Applications
[5] ETSI ES 201 873-1: Methods for Testing and Specification (MTS): he Testing and Test Control notation version 3; Part 1: TTCN-3 Core Language
[6] I. Schieferdecker and B. Stepien, Automated testing of XML/SOAP based web services
[7] P.Deussen, G. Din, I. Schieferdecker, A TTCN-3 based online test and validation platform for internet services.
[8] H.Q. Nguyen, B. Johnson, M. Hackett, Testing Applications on the Web, 2nd Edition, Wiley, 2003