Integration Testing of Web Applications and Databases using TTCN-3

Bernard Stepien, Liam Peyton

School of Information Technology and Engineering,
University of Ottawa, Canada
{bernard, lpeyton}@site.uottawa.ca

Abstract. Traditional approaches to integration testing typically use a variety of different test tools (such as HTTPUnit, Junit, DBUnit) and manage data in a variety of formats (HTML, Java, SQL) in order to verify web application state at different points in the architecture of a web application. Managing test campaigns across these different tools and correlating intermediate results in different formats is a difficult problem which we address in this paper. In particular, the major contribution of this paper is to demonstrate that a specification-based approach to integration testing enables one to define integration test campaigns more succinctly and efficiently in a single language/tool and correlate intermediate results in a single data format. We also evaluate the effectiveness of TTCN-3 (a standards-based test specification language and framework) in supporting such an approach.

Keywords: web applications, integration testing, databases, TTCN-3

1 Introduction

Complex web applications in a service-oriented architecture may have to integrate data from several data sources and may have to maintain state in a distributed fashion across many components of the web application. One of the aims of integration testing is to verify intermediate results at key interaction points within the architecture of the web application. This can be done by testing the web application state as captured either in persistent data stored in a data source, or as session data maintained in memory by the web application or any of its distributed components. This can be a complex and challenging task even under the most ideal circumstances.

Traditional approaches to integration testing would typically use a variety of different test tools (such as HTTPUnit, Junit, DBUnit) and manage data in a variety of formats (HTML, Java, SQL) in order to verify web application state at different points in the architecture of a web application. Managing test campaigns across these different tools and correlating intermediate results in different formats is a difficult problem which we address in this paper. In particular, the major contribution of this paper is to demonstrate that a specification-based approach to integration testing enables one to define integration test campaigns more succinctly and efficiently in a single language/tool and correlate intermediate results in a single data format. We

also evaluate the effectiveness of TTCN-3 (a standards-based test specification language and framework [1]) in supporting such an approach.

Using the TTCN-3 specification language, we define an abstract data layer which can maintain web application state across a variety of test tools and formats and verify intermediate results based on tests which transform that abstract data layer. The approach is implemented in a TTCN-3 test framework which uses a collection of test adaptors to mediate between the abstract test layer in which test specifications are defined and the concrete test layer which interacts directly with the web application and its components. Specific examples based on an Online Book Store and sample TTCN-3 specifications are used to verify and correlate database behavior and web application component behavior as they relate to web application state.

TTCN-3 is a test specification and test implementation language for testing distributed systems developed by the European Telecommunications Standards Institute (ETSI). It provides powerful abstraction mechanisms for interfacing to different data and presentation formats and for defining test cases at different levels of abstraction, much as developers use modeling languages to specify the design of a system at different levels of abstraction. This enables reuse across different levels of test activities [2] and the coordination and synchronization of test activities with development activities throughout the development life cycle.

The need for a systematic test framework reflective of web application architecture rather than a patchwork of tools and test scripts has been pointed out as well in other work [3] outside of the TTCN-3 community. Other approaches to integration testing have focused on ensuring formal conformance to web service protocols in web applications that leverage web services as components [4]. TTCN-3 has also been used in this manner [5].

An alternative approach taken to address the low level of detail at which current tools operate is to do model-based testing where test scripts are generated from models. This was done in the AGEDIS case studies [6] where HTTPUnit and HTMLUnit scripts were generated from UML models. In [7] User Requirements Notation (URN), an ITU standard for requirements modeling in telecommunications was used to test web applications. And in [8] evaluations done with JML-JUnit used JUnit scripts generated from JML models of Java classes. Such approaches do link test script generation to an abstract view of the system being tested, but they do not give the same power and flexibility as a test specification approach to verify application logic and information management independent of volatile implementation and presentation details.

2 Book Store Web Application Example

Figure 1 gives a simple example of a typical J2EE web application that supports an on-line book store. We will use this example, throughout the paper to illustrate our approach. The browser interface contains a rich set of HTML, XML, JavaScript, images, stylesheets, etc, that it receives from the web application in response to HTTP requests. The web application, in return, interacts with a variety of components within a service-oriented architecture. It interacts with the book order database via

JDBC to keep track of available books and purchases. It interacts with a shopping cart enterprise java bean via RMI while the customer is shopping online and it interacts with an order processing service via SOAP to let the warehouse know when there is an order of books to ship.

There is also a test framework (implemented in TTCN-3) which can communicate directly via concrete test adaptors with either the web application or any of the components used by the web application. It performs integration testing that verifies intermediate results in terms of application web state based on abstract test specifications which define an abstract data layer in terms of data types, and uses templates for expected responses.

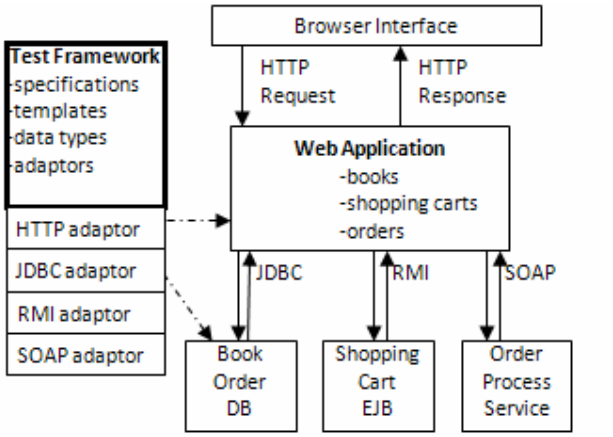

Figure 1 – Book Store and Test Agent

For the purposes of explaining the specification-based approach to integration testing, we will focus on first verifying intermediate results in the Book Order DB, and then on how to correlate and integrate those results into verification of intermediate results within the web application itself.

3. Database Integration Testing

Specifying test suites for testing database integration mostly consists of specifying test oracles for results of queries or test oracles to verify the state of a database after some update operation. In both cases, testing consists in performing a query and verifying the result set. The central TTCN-3 concept to represent test oracles is the template. In sharp contrast with traditional testing methodologies, a TTCN-3 template performs the checking of all the data involved in a result set in a single operation. The setting of an oracle is organized in three steps:

- The definition of a data type to represent the results set
- The definition of a template of values for each element of the data type to be matched.

- The definition of a test events sequence with possible alternative paths.

In contrast to unit testing approaches, TTCN-3 is more behavior oriented. This includes the results of several consecutive events where the current state of the database is dependent on all previous steps.

3.1 Data typing

Abstract data types to represent database results are defined in terms of results sets as sets of rows where each row is an individual database record and columns correspond to fields. Union in TTCN-3 can be used to define results that join elements from different tables. For example, below we define a BooksTable, PublisherTable and the join between them to create a CatalogEntry result set.

```
type record DBBooksTableType {
   charstring author,
   charstring title,
   float price
}

type record DBPublisherTableType {
   charstring pubName,
   charstring street,
   charstring city
}

type record BooksPublisherResultType {
   charstring pubName,
   charstring author,
   charstring title
}
```

The three above basic types are merged into a TTCN-3 union type as follows:

```
type union CatalogEntry {
   DBBooksTableType booksTable,
   DBPublisherTableType publisherTable,
   BooksPublisherResultType bookspubResult
}
```

Finally, rows are represented using the TTCN-3 set of type construct on the previously defined union.

```
type set of CatalogEntry CatalogEntryResultSet;
```

3.2 Specifying test oracles using TTCN-3 templates

The TTCN-3 template is more than an instance of test data for a given data type. It looks like a plain assignment of values but is in fact capable of complex matching mechanism. Values can instead be list of alternate values, ranges for numeric values

or pattern matching specifications for strings. Thus, a TTCN-3 template is more like a hybrid between a data assignment and some potentially complex logical expression. Since it can be parametric it actually serves the role of a function. The most important concept however remains that the actual matching mechanism is built-in and thus needs zero coding effort from the tester. An atomic element of a test oracle can be represented by the following template where the field `booksTable` indicates the type among the types of the union used and the assigned values indicate the orcale's criteria.

```
template CatalogEntry amerique := {
   booksTable := {
      author := "Herge",
      title := "Tintin en Amerique",
      price := 8.20
   }
}
```

The template is also a powerful structuring concept since it can be re-used by other templates. For example, a list of database items defined individually as above can be re-used to define a list of items, thus different database rows, as follows:

```
template CatalogEntryResultSet myBooks := { templeSoleil, ileNoire, amerique };
```

Zoio recommends to externalize assertions [9] by avoiding the scattering of hard coded assertions throughout the test code and also to write comprehensive tests that cover all aspects of a database state and finally use precise assertions. The TTCN-3 template concept naturally implements all of these recommendations.

3.3 Performing a test

In TTCN-3, a database test is specified by sending an SQL request over a communication channel that is abstracted as a TTCN-3 port and by receiving a response over that same channel. The receive statement actually serves two purposes: to actually obtain data from the communication channel and to match this obtained data with a template. In the following example, after sending an SQL request, we attempt to match the result set to the template `myBooks` that we defined previously.

```
db_port.send("select * from books");
alt {
     [] db_port.receive(myBooks) {setverdict(pass)}
     [] db_port.receive {setverdict(fail)}
}
```

The above code illustrates how test verdicts are set by using the TTCN-3 `alt` construct. The first case corresponds to the expected response being received and the test verdict being set to pass. The second alternative case consists in receiving anything else which would result in setting the test verdict to fail instead. The TTCN-3 alt construct is a powerful concept that enables one to specify complex test

behaviors through nesting as trees that represent various possible sequences of test events with their corresponding test verdicts in the leafs of the tree.

3.4 Separation of concerns between abstract and concrete layer

So far we have only defined a high level abstract test specification without specifying how the data is actually obtained. This is because TTCN-3's central concept is the separation of concern between the abstract and concrete layers. The concrete layer called the test adapter layer is where the actual connection with a database occurs and where the data is retrieved from results set specified in a general purpose language (GPL) such as Java depending on the implementation language of the TTCN-3 tool used. However, the actual classes and member functions for the test adapter are fully defined in the TTCN-3 standard part 5 [10]. There is a correspondence between the abstract layer `send` command and the test adapter's `triSend` method. This is where the typical JDBC [11] database interface would take place and, at this point, nothing is unusual compared to the traditional GPL test implementation, as can be observed in the following example of the implementation of the `triSend` method.

```
public TriStatus triSend(final TriComponentId componentId,
      final TriPortId tsiPortId, final TriAddress address,
                           final TriMessage sendMessage) {

   byte [] mesg = sendMessage.getEncodedMessage();
   if(tsiPortId.getPortName().equals("system_dbPort")) {
      String theSQLRequest = new String(mesg);
      Connection db_connection = null;

      try {
         Class.forName ("com.mysql.jdbc.Driver").newInstance ();
      } catch …

      try {
         String url = "jdbc:mysql://localhost/ebookstore";
         db_connection = DriverManager.getConnection (url, null, null);
      } catch (SQLException e) { … }

      try {
        Statement db_statement = db_connection.createStatement();
        boolean status = db_statement.execute(theSQLRequest);
      } catch …

      ResultSet theCurrentResultsSet  = db_statement.getResultSet();
}  }
```

The results of the statement's execution would then be retrieved and transformed into abstract data by a codec (coder/decoder) defined in the test adaptor. For that purpose, the result set object instance is serialized so that it can be passed as a byte[] stream to the codec.

```
  byte[] theByteRepresentation = ((DBCodec)
         getCodec("")).serializeObject(theCurrentResultsSet);
```

The codec can be written in different ways. Normally, there is a corresponding codec for each abstract data type. For JDBC, we have found a more generic approach for a codec that can handle any abstract data type without having to know what data types are used in a test suite. Thus, this codec is a perfect framework that can be used in any database testing application. Its full description can be found in [12]. The separation of concerns of TTCN-3 has some additional benefits of re-usability of the abstract layer across platforms and implementation languages.

4 Integration testing of web and database applications

So far, we have shown examples of test specifications that, despite their abstractedness, are not too different from unit testing since they involve only the database. The need to test databases in conjunction with the web application that uses them has been pointed out in [13]. They report on a tool called AGENDA that produces test paths using a cyclomatic complexity algorithm. It is based on a white box approach and addresses three concerns: better coverage, more appropriate input values for forms and better targeting of test efforts. However, they use plain XML files to assemble their test specification which unfortunately adds some unnecessary complexity to the problem.

The real value of using TTCN-3 is beyond mimicking unit testing and instead is found in the specification of complex systems that consist of various components that perform different services, some being database services and others being web services or user interface services such as presenting web pages. Combining such composite services into a single integration test can be challenging when using a GPL. This is mostly due to the frequent tendency to mix test assertions and data extraction functionalities. The separation of concerns that TTCN-3 supports enables us to specify test suites strictly at the abstract level and thus enable the tester to focus on the purpose of the test.

A frequent class of applications consists in the combination of web applications with databases. Here, two kinds of tests can be performed:

- Check the database state after a web user submitted data through a web application.
- Check the results after a user did a query to the database over a web application to see if they correspond to the state of a database.

4.1 Consistency check between web data entries and database state

Web data entry is achieved by submitting web forms that have been filled with data. Thus, in order to specify the test to check the consistency between the web data entries and the resulting database state, we need to handle both aspects of the integration test and first how to submit a form in an abstract way and eventually how to translate this abstract request into a concrete web query.

A complete description of various approaches to achieve the above has been presented elseqhere [14]. Here we will briefly show the essential abstract layer

elements required to specify an HTML form so as to be able to illustrate the concept of test oracle transformation later.

```
type record ParameterValuesType {
   charstring parmName,
   charstring parmValue
}

type set of ParameterValuesType ParameterValuesSetType;

type record FormSubmitType {
   charstring formName,
   charstring buttonName,
   charstring actionValue,
   ParameterValuesSetType parameterValues
}
```

Using the above abstract data type, we can specify TTCN-3 templates for web form submissions. First a definition of a filled web form for entering a specific book:

```
template ParameterValuesSetType filledFormAmerique := {
   {parmName := "author", parmValue := "Herge"},
   {parmName := "title", parmValue := "Tintin en Amerique"},
   {parmName := "price", parmValue := "8.00"}
}
```

Then we specify a parametric template to describe the form itself using a formal parameter to indicate the actual form parameters values for a specific book. This template can be re-used to submit an arbitrary number of different books.

```
template FormSubmitType webInsertionFormSubmit
            (ParameterValuesSetType theParameters) := {
   formName := "bookAdditionForm",
   buttonName := "add",
   actionValue :=
        "http://localhost:8080/eBookStore/servlet/book_insertion",
   parameterValues := theParameters
}
```

Finally, we specify the typical test behavior statement that executes this form submission, namely a send command with the parametric template fully instantiated with the previously defined template about the elements of the book being inserted in the database and finally a receive statement that attempts matching the web response to yet another template defining the expected web response.

```
web_port.send(webInsertionFormSubmit(filledFormAmerique));
web_port.receive(webResponsePage);
```

The test adapter layer's codec would then produce the appropriate web request as follows:

```
http://localhost:8080/eBookStore/servlet/book_insertion?&author=Herge&ti
tle= Tintin en Amerique&price=8.00
```

This web request would then be submitted on a TCP/IP channel using a post command. This example also illustrates how the TTCN-3 template achieves another separation of concern between test behavior and conditions governing behavior.

At this point we have successfully submitted the filled form and all we need to do is to perform a test on the database to see if the data has been stored using the test described in section 2. However, this would be a kind of double hard coded test oracle approach. We certainly cannot avoid hard-coding the form submission since we need some starting point; we could, however, avoid hard-coding the test oracle for the database results by merely transforming the form submission template into a database result set template to check the state of the database. This is possible in TTCN-3 because of its ability to specify dynamic templates that are constructed from other tests results. The most important fact here is that with TTCN-3 we can do such transformation without having all the data encoding or extraction that would be required in a GPL. Thus, this transformation can be achieved relatively concisely at the abstract level as in the following example:

```
function transformForms2DB(FormsParametersValuesSetType theFormParms)
return CatalogEntryResultSet {
   …
   for(i:=0; i < numOfForms; i:=i+1) {
      anItem.booksTable.author :=
               getFieldValue("author",theFormParms[i]);
      anItem.booksTable.title :=
               getFieldValue("title", theFormParms[i]);
      anItem.booksTable.price :=
               str2float(getFieldValue("price", theFormParms[i]));
      theItems[i] := anItem;
   }
   return theItems;
}
```

Thus, a complete integration test can now be specified as follows:

```
testcase web2DatabaseResultsTest() runs on MTCType system SystemType {
   var DBSelectResponseType theDBSelectResponse;

   map(mtc:dbPort, system:system_dbPort);
   map(mtc:webPort, system:system_webPort);

   // database re-initialization
   dbPort.send("delete from books");

   // have a user insert a book through a web page form
   webPort.send(webInsertionFormSubmit(filledFormAmerique));
   …
   // transform the list of filled forms information into
   // a database query results template
   var CatalogEntryResultSet expectedDatabaseResults :=
          transformForms2DB({filledFormOrNoir, filledFormAmerique});

   // check if the database contains the entered books
   dbPort.send(myBooksSelectRequest);
```

```
   alt {
      [] dbPort.receive(myBooksSelectResponse(expectedDatabaseResults))
       {
          setverdict(pass)
       }
      [] dbPort.receive {
          setverdict(inconc);
       }
}  }
```

4.2 Consistency check between database state and web queries

Given a specific state of the database, we define a test that consists in simulating a user performing a query through a web page and obtaining data that is displayed on the response page. The second step of the test consists in performing a direct SQL database query to obtain the same data as through the web page and compare it to the data obtained through the web page. If the two sources of data coincide, the test has passed.

The second step of this test is identical to the second step of the previous test (web data insertion against database query). We can re-use the same SQL statement for that purpose. These SQL statements can be extracted from the application under test as suggested in [15]. They propose a testing approach that transforms the embedded SQL statements in database applications to procedures in a general-purpose programming language (GPL). Here we replace the GPL with TTCN-3 and gain clarity and conciseness. The first step however is somewhat similar since we need to submit a form with some pre-filled fields, this time with the parameters of the query and with the different requested actions as follows:

```
template FormSubmitType queryBooksHerge := {
   formName := "queryForm",
   buttonName := "query",
   actionValue :=
        "http://localhost:8080/eBookStore/servlet/book_selection",
   parameterValues := {
     {parmName := "author", parmValue := "Herge"},
     {parmName := "maxPrice", parmValue := "10.0"}
   }
}
```

Again, this web query is submitted to the web application using a TTCN-3 send command as follows:

```
webPort.send(queryBooksHerge);
```

This web query will result in a web response page that we need to specify using TTCN-3 abstract data types and templates. A full description on how to achieve this can be found elsewhere [14]. Here we summarize some main ideas. A web page is modeled using the following types:

```
type record WebPageType {
```

```
    integer statusCode,
    charstring title,
    charstring content,
    LinkListType links optional,
    FormSetType forms optional,
    TableSetType tables optional
}
```

Web page data is typically displayed using HTML tables that can be modeled with the following TTCN-3 types:

```
type set of charstring RowCellSetType;
type record TableRowType {
   RowCellSetType cells
}
type set of TableRowType TableRowSetType;
type record TableType {
   TableRowSetType rows
}
type set of TableType TableSetType;
```

Once the types are defined, we can define the parametric template for the web response that is composed of constants such as the page title and the status and a parameter for the actual tables containing the requested data.

```
template WebPageType
            hergeDBQueryResultsPage(TableSetType theTables) := {
   statusCode := 200,
   title := "bookstore.com query items page results",
   content := ?,
   links := {},
   forms := {},
   tables := theTables
}
```

Here again, we could have hard coded the values of the tables but, instead, in order to avoid duplicate work we prefer to dynamically create it by deriving it from the result set of the database query using a function as follows:

```
function transformDBResultsIntoHTMLTables(ItemsType theDBItems)
                                                  return TableSetType {
   …
   theTableRows[0] := { cells := {"author", "title", "price" }};

   for(i:=0; i < numOfDBRows; i:=i+1) {
      if(ischosen(theDBItems[i].booksTable)) {
         aBook := theDBItems[i].booksTable;
         aRow := {
            cells := { aBook.author, aBook.title,
                                          myFloat2str(aBook.price) }
         };
         theTableRows[i+1] := aRow;
     }
   }

   theTable := { rows := theTableRows };
```

```
   tables[0] := theTable;
   return tables
}
```

Finally the full test behavior is specified as follows:

```
testcase database2webResultsTest() runs on MTCType system SystemType {
   var DBSelectResponseType theDBSelectResponse;

   map(mtc:dbPort, system:system_dbPort);
   map(mtc:webPort, system:system_webPort);

   … // set the database in the desired state

   dbPort.send(myBooksSelectRequest);
   dbPort.receive(myBooksSelectResponse(myBooks))
                                -> value theDBSelectResponse {
   var CatalogEntryResultSet theReceiveDBItems :=
                                      theDBSelectResponse.items;

   var TableSetType booksTables :=
          transformDBResultsIntoHTMLTables(theReceiveDBItems);

   webPort.send(queryBooksHerge);
   alt {
      [] webPort.receive(hergeDBQueryResultsPage(booksTables)) {
            setverdict(pass)
      }
      [] webPort.receive {
            setverdict(fail)
      }
   }
}
```

5 Conclusions and Future Work

In this paper, we have demonstrated two main advantages of a test specification approach for integration testing. First, test cases can be defined much more succinctly using a single common language. This simplifies not only the writing of test cases, but also the reading and understanding of these test cases. It also eliminates the need to consult and understand test cases defined and written in several different languages. Secondly, and perhaps more importantly it enables intermediate results that are communicated using different data formats and protocols, to be integrated, combined, compared and verified within a single, consistent data abstract layer.

We have also demonstrated the suitability of TTCN-3 both as a test specification language and as a framework for executing integration tests. It supports the definition of an abstract specification layer separate from test adaptors which manage implementation specifics. TTCN-3 templates that are used to specify test oracles are created dynamically based on defined abstract transformations between web requests and the virtual data layer. The virtual data layer is mapped to different database tables or views by a universal data codec.

While we have focused on integration testing in this paper, it can also be used for blackbox and white box testing related to databases and session state. Black box testing using parallel testing is proposed in [16]. They particularly recommend to avoid the traditional approach of resetting the state of a database before each test as is often recommended [9] because this is a time consuming process and also because it does not reflect the realities of a multi-user application in general. In TTCN-3, we have already shown the benefits of multi-user application testing in [17] and believe the extension of these principles to databases should be straightforward.

Whitebox testing as described in [18] can also be implemented in a straight forward fashion at an abstract level using TTCN-3. They state that the full behavior of a database application program is described in terms of the manipulation of two very different kinds of state: the program state and the database state. While, so far, we have used a message oriented approach in our abstract test suites, TTCN-3 provides also a procedure oriented approach. It can be used to invoke functions or methods of the application under test directly and, thus, check the resulting state of both the software and the database.

5 Acknowledgements

The authors would like to thank Testing Technologies IST GmbH for providing us the necessary tool -- TTworkbench -- to carry out this research as well as NSERC for partially funding this work.

References

1. ETSI ES 201 873-1, The Testing and Test Control Notation version 3, Part1: TTCN-3 Core notation, V3.4.1, September 2008
2. R. L. Probert, P. Xiong, B. Stepien, "Life-cycle E-Commerce Testing with OO-TTCN-3", FORTE'04 Workshops proceedings, September 2004
3. C.Rankin, The Software Testing Automation framework, IBM Systems Journal, Software Testing and Verification, Vol. 41, No.1, 2002
4. A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of web services for testing conformance to open specified protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, Architecting Systems with Trustworthy Components, number 3938 in LNCS. Springer-Verlag, 2006.
5. B.Stepien, I.Schieferdecker, "Automated Testing of XML/SOAP based Web Services", Proc. of the 13th. Fachkonferenz der Gesellschaft f¨ur Informatik (GI) Fachgruppe KiVS, February 2003
6. Craggs I., Sardis M., and Heuillard T. AGEDIS Case Studies: Model-based Testing in Industry. Proc. 1st European Conf. on Model Driven Softw. Eng. (Nuremberg, Germany, Dec. 2003), imbus AG, 106—117
7. D. Amyot, J-F Roy, M. Weiss, UCM-Driven Testing of Web Applications. SDL Forum 2005
8. R.P.Tan, S.H. Edwards, Experiences Evaluating the Effectiveness of JML-JUnit Testing, ACM SIGSOFT Software Engineering Notes, September 2004 Volume 29 Number 5

9. P. Zoio, “Testing 1,2,3…”, Oracle Magazine, July-August, 2005. http://www.oracle.com/technology/oramag/oracle/05-jul/o45testing.html
10. ETSI ES 201 873-5 V3.3.1, The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI), April 2008
11. JDBC, http://java.sun.com/docs/books/tutorial/jdbc/index.html
12. B.Stepien, A generic TTCN-3 codec framework for testing Database applications, Working Paper, School of Information Technology and Engineering, University of Ottawa, 2008.
13. Y. Deng, P. Frankl, J. Wang, Testing Web Database Applications, ACM SIGSOFT Software Engineering Notes, Volume 29 , Issue 5, pp 1-10, 2004.
14. B. Stepien, L. Peyton, P. Xiong, “Framework Testing of Web Applications using TTCN-3”, International Journal on Software Tools for Technology Transfer, Springer Berlin / Heidelberg. Vol. 10, No. 4, pp 371-381, 2008
15. M.Y. Chan, S.C. Cheung, Testing Database Applications with SQL Semantics, in Proceedings of 2nd International Symposium on Cooperative Database Systems for Advanced Applications (CODAS'99), 1999.
16. C. Binnig, D. Kossmann, E. Lo, Testing Database Applications in Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 206.
17. L. Peyton, B. Stepien, P. Seguin, "Integration Testing of Composite Applications", Proceedings of the 41st Hawaii International Conference on System Sciences (HICSS 2008), 2008. ISSN:1530-1605. http://csdl.computer.org/comp/proceedings/hicss/2008/3075/00/30750096.pdf
18. D. Willmor, S. M Embury, Exploring test adequacy for database systems, Proceedings of the 3rd UK Software Testing Research Workshop, September 2005