CONFORMIQ

A Conformiq Technology Brief

Testing Bluetooth® Protocol Stacks with Computer-Generated Tests

Bluetooth is an ubiquitous, open wireless technology standard for exchanging data over short distances. A standard originally created by Ericsson and now managed by the Bluetooth Special Interest Group, Bluetooth has become an indispensable part of the global digital communications fabric. Implementations of Bluetooth are hardware-based and subject to stringent quality requirements. Because of their nature, recalling or updating malfunctioning Bluetooth chips is cumbersome and costly. Furthermore, a low-quality Bluetooth implementation can be costly to brand value; Internet forums are filled with complaints about malfunctioning Bluetooth devices that do not interoperate with each others—searching for "bluetooth connection problem" results in more than 300,000 hits on Google. These factors underline the importance of proper and thorough functional testing of Bluetooth implementations, and in particular the importance of designing test cases that cover both common as well as corner case situations. In this Conformiq Technology Brief we use Bluetooth as an example of a domain where the test design problem can be successfully solved with Conformiq Automated Test Design™, a methodology that improves test design productivity and quality by automating parts of the test design process.

The two crucial parts of any functional testing process are *test design* and *test execution*. Test design is about deciding what to test and how, selecting input data, and defining test oracles (verification conditions). Test execution is about actually running the tests (manually or, more preferably, in an automated fashion) and logging the results. Recently, companies have put a lot of focus on improving and automating test execution, but actually it is test *design* that drives product quality and time to market. Test design is usually manual (that is, tests are designed by humans); this is so commonplace that often test managers and quality assurance directors do not even actively perceive it as process that *could* be automated.

To illustrate the practical process of moving to automated test design—computer-generated functional tests—we consider in this brief the problem of designing tests for some core features of a Bluetooth protocol stack implementation.

Copyright © Conformiq Inc. 2010. All Rights Reserved.

Scope

In the basic scenario, one Bluetooth-enabled device wants to connect with another one. The device initiating the connection is called the client; the device that is waiting for a connection attempt is known as the server. In order for the client to be able to connect to the server, it needs to be able to search for servers and to enquire them for attributes. We consider these two features in this brief (*service search* and *service attribute request*) and a third feature which combines these two (s*ervice search for attribute*).

The Bluetooth protocol stack as a whole contains many other functions. We chose these three features as the scope of this brief, but the same approach could be extended to other aspects of the Bluetooth specifications as well as to other protocols, other reactive control systems, and to any other business- or mission-critical software components, system functions or systems.

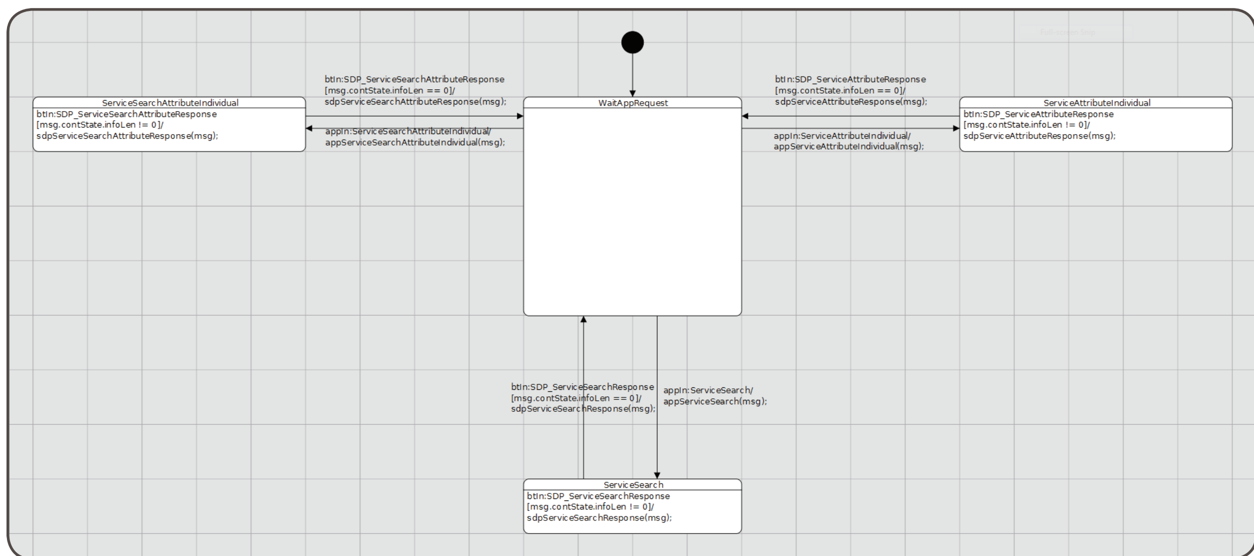
Creating a Computer-Readable Specification

In order for us to be able to use a computer to generate tests for a Bluetooth stack implementation, we first need to put the stack's relevant specifications into a format that a computer can understand. In the Conformiq approach we call these computer-readable specifications *models* as they *model* the actual system (this is where the term *model-based testing* comes from).

The models that Conformiq Designer™, our test generation tool, can understand consist of state chart diagrams as well as code in a Java-based programming language (for an example from our model, see the last page). Engineers use these diagrams and code to describe the correct, intended operation of the system under test, on a chosen level of abstraction.

For the Bluetooth features considered here, the model we have created consists of two state charts and some program code. The state charts describe the high-level control flow and the program code describes the details about data in the fields of Bluetooth protocol packets.

One of the two state charts is shown below. This state chart describes three possible control flows for the protocol stack. The central state WaitAppRequest denotes those states of the protocol stack where the stack is waiting for the application (e.g., the user interface of a mobile phone) to initiate a connection attempt. An initiated connection attempt is represented by a ServiceSearch message that comes from the application via an interface named appIn in the model; the ServiceSearch message is processed on the arrow going downwards from the WaitAppRequest state. This means that when the application requests a service search, the protocol stack moves to the ServiceSearch state (bottom state in the diagram). In this



One of the two state charts in our Bluetooth model

state, the stack is collecting service search responses from the Bluetooth server. Eventually the server indicates "no more" responses, which then triggers transition from the ServiceSearch state back to the WaitAppRequest state (arrow pointing upwards).

In general, in our model the Bluetooth implementation resides between an application (which wants to use Bluetooth) and the server. What the model describes is that the application can request different types of actions from the Bluetooth implementation, which then communicates with the external Bluetooth server. The key here is that the Bluetooth implementation is modeled as having two external interfaces (to the application and to the server), and our approach generates test cases that drive both those interfaces simultaneously, removing need to do manual stubbing or to use simulators to close the circuit—and all this without having to specify or know how the application or the server actually behaves, as their necessary behaviors (which correspond to test inputs to the bluetooth implementation) are *generated automatically*.

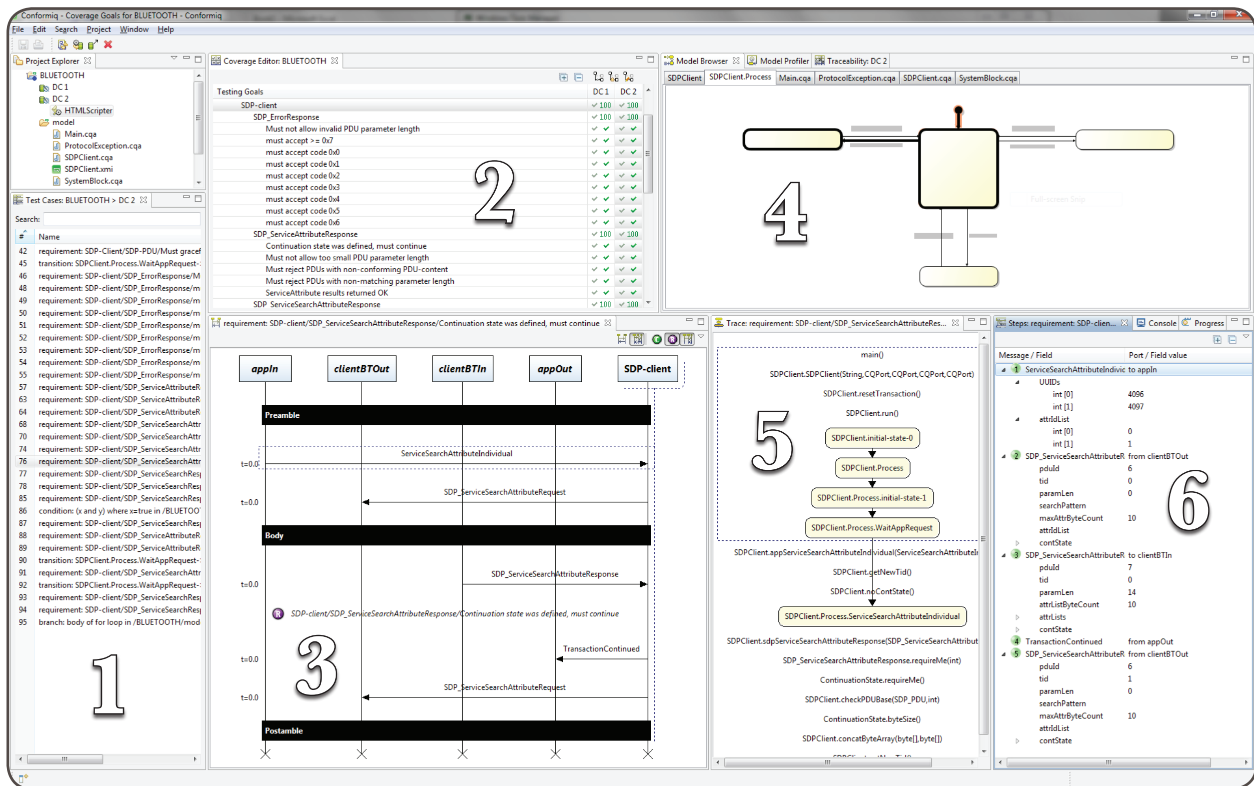
Using Conformiq Designer

Conformiq Designer is our Automated Test Design™ tool. It can read in models for test generation from a variety of tools, and it provides a graphical, interactive workflow for automatic test generation. The screenshot below illustrates some of the key features and views:

1. Test set view—our tool generates test cases automatically based on a model describing the correct operation (*system model driven* test generation) and manages the generated test sets.

2. Coverage editor—the user has complete control on setting or blocking test goals related to both human-defined requirements as well as out-of-the-box testing heuristics such as boundary value analysis or transition coverage.

3. Graphical test case display—generated test cases can be viewed graphically as message sequence charts (MSCs) which illustrate the flow of a test



Conformiq Designer IDE for automatic test case generation

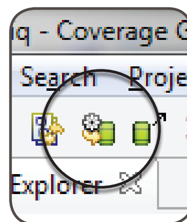
case especially well in the case of multi-component and multi-interface testing scenarios.

4. Graphical model view—regardless of the tool from which the model has been imported, Conformiq Designer can display the graphical structure of the model and highlight those parts of the model which are covered by any individual test case or test step.

5. Detailed step view—an alternative view that illustrates how a single test case relates to the behavior of the model.

6. Test data view—Conformiq Designer generates all test data automatically, both inputs as well as expected outputs. The test data view allows the user to review the generated test data in a hierarchical tree display.

All these views let the user to examine the results of automatic test generation; the test generation process itself is fully automatic and triggered by the push of a button.



Test Generation by Symbolic Exploration

Conformiq Designer is based on a highly specialized variant of the general idea of test generation by *symbolic state-space exploration*. This is the state-of-the-art method for automatic test generation from models that describe the behavior of the system under test.

It can be mentioned that many other automatic test generation methods exist, for example generating pairwisely combined test data using combinatorial methods (Taguchi designs; covering arrays) or generating opaque test sequences by the Chinese Postman algorithm. These have, however, a narrow scope of applicability and cannot be used to automatically solve test design problems on the level of complexity of general software components or systems—such as a Bluetooth stack.

Results of Test Case Generation

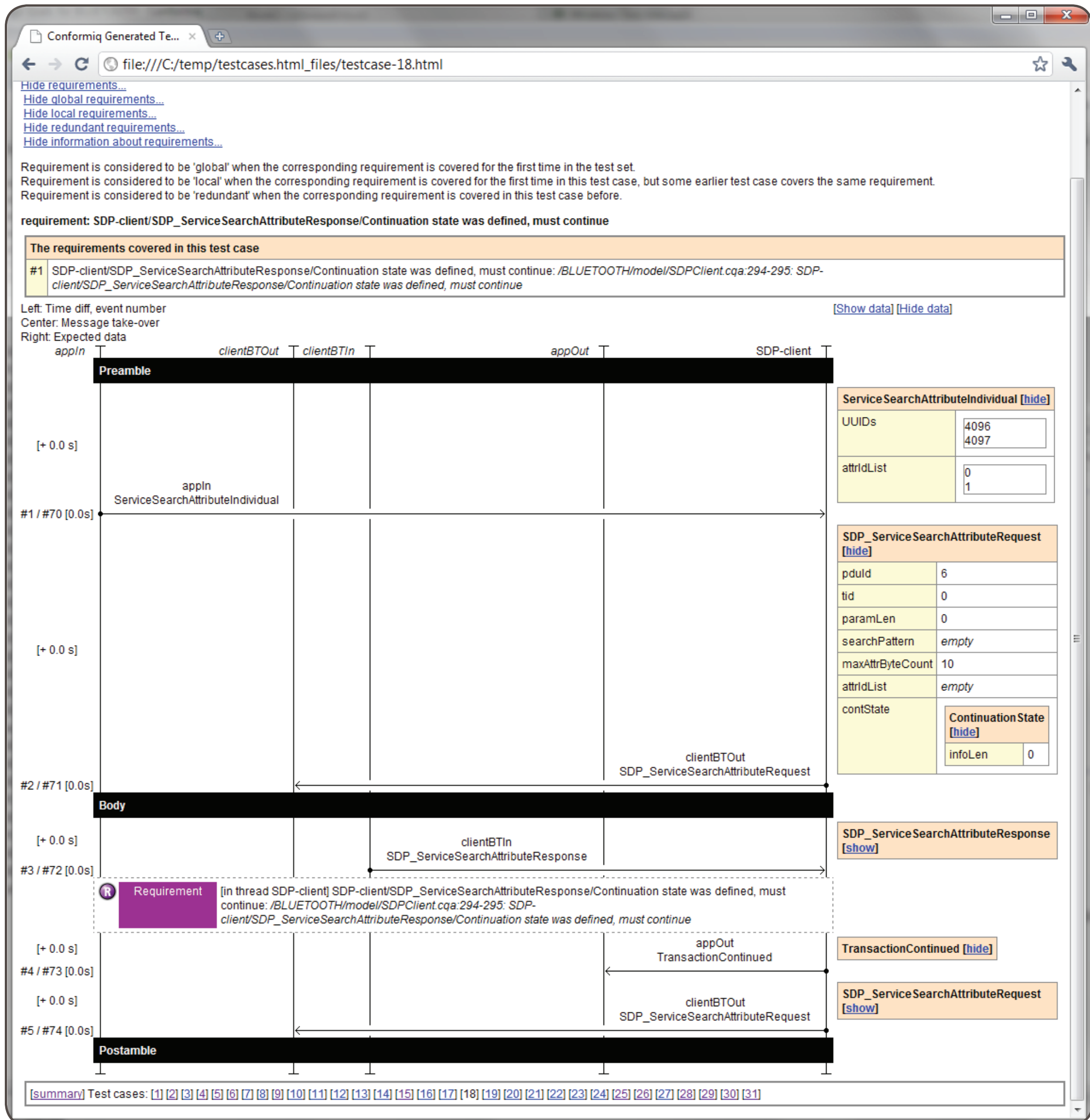
In addition to generating the actual tests, Conformiq Designer generates also auxiliary documentation such as traceability data (to trace the generated test cases back to human-defined requirements) and test case dependency data. The generated dependency data be used to optimize test case execution when some of the test cases fail: if a test case fails, the test cases that depend on it can be skipped in execution.

The actual test cases can be output in various formats and the formats can be completely user-defined. One popular option is to output the test cases both in an executable format (such as Perl, Python or TTCN-3 scripts) and at the same time in a human-readable format, such as HTML (web) pages. On the next page there is a screenshot showing one of the generated test cases for Bluetooth as a HTML document displayed within a web browser, complete with a graphical message sequence chart view as well as hierarchical test data (shown on the right) and requirements traceability markup (the purple annotation in the lower part of the screen).

One benefit of being able to export HTML and other human radable formats from the tool is that this makes it possible to review the test cases even by team members who do not use Conformiq Designer by themselves.

So if we look at the test case on the next page, it is easy to understand its structure even if one is not familiar with Bluetooth. There are four communication points on the external interface of the system under test: application input (data from application) and application output (data to application); and Bluetooth protocol input (data from server) and Bluetooth protocol output (data to server). These correspond to the vertical lines appIn, appOut, clientBTIn and clientBTOut, respectively.

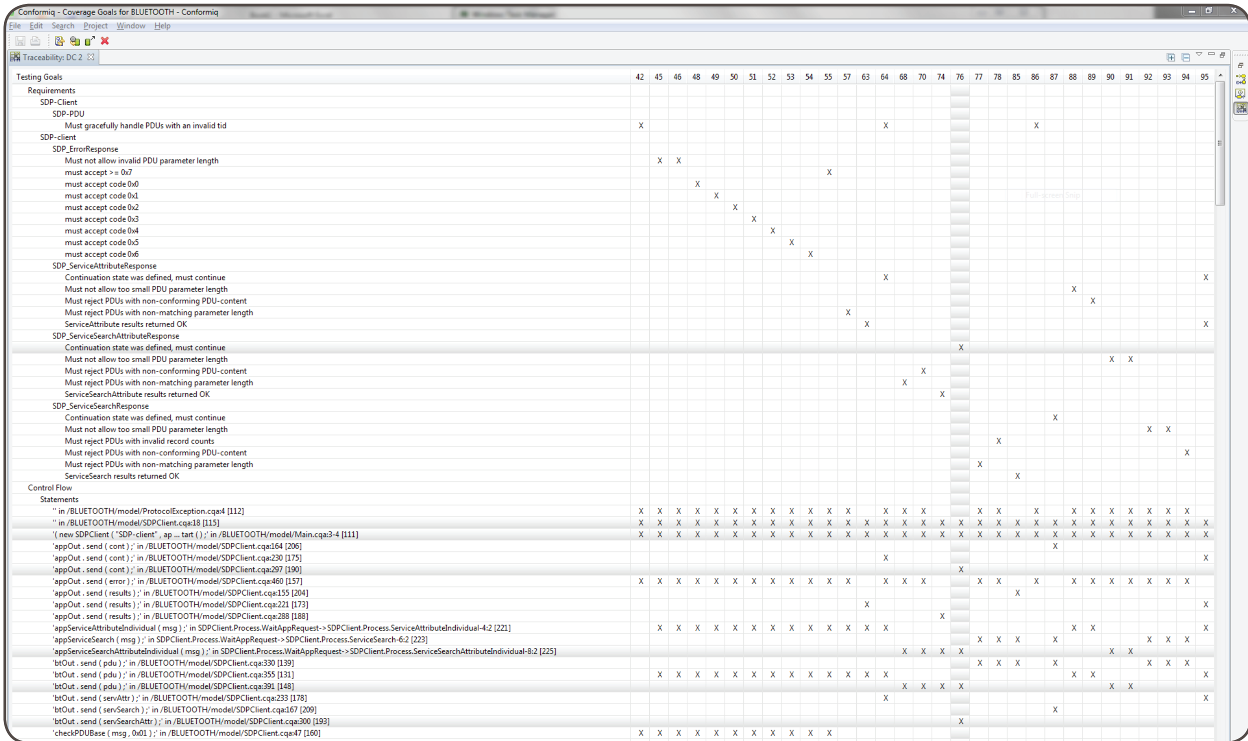
The test case starts by a request from the application to the Bluetooth implementation to search for a service and some related attributes; this is shown as an arrow from appIn to SDP-client, i.e. an input to the system under test. It is natural that the test case starts by an input to trigger the Bluetooth implementation to begin a transaction.



One of the generated test cases rendered into HTML and displayed within a browser for human review

What next happens in the test case is that the SDP client is *assumed* to send out a ServiceSearchAttributeRequest message to the Bluetooth server. This is shown as an arrow from SDP-client to clientBTOut. During test execution this is an expected output to be verified. The third message comes then to the system under test from clientBTIn, i.e. it is a simulated response from the external Bluetooth server. Note that in order to execute the test cases, not actual Bluetooth server implementation is needed, as the response that is required to drive the client forwards is synthesized by Conformiq Designer. (Test data for the last three messages is not shown to fit the diagram on one page.)

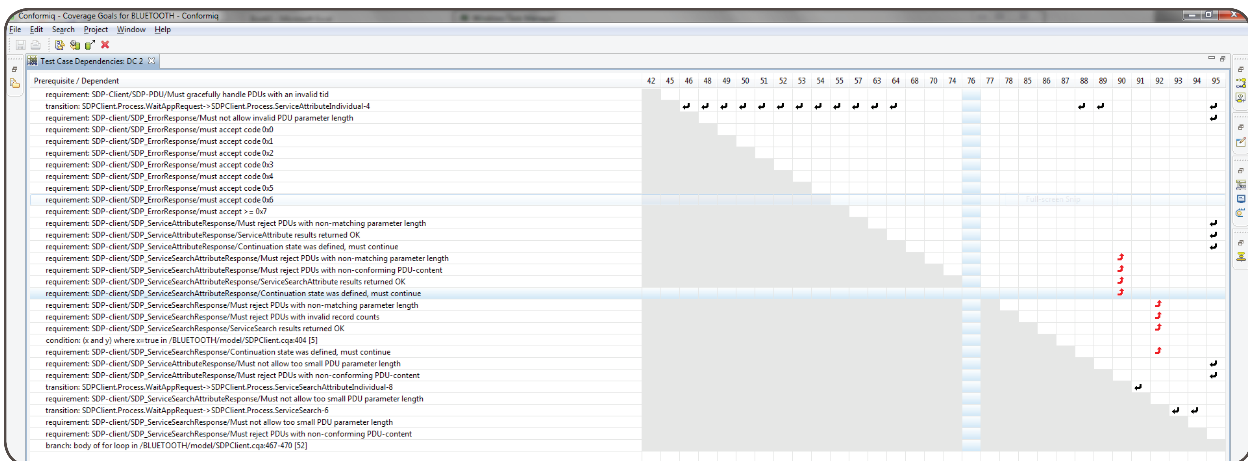
The purple "Requirement" annotation highlights that at this point the crux of this computer-generated test case is reached: the purpose of the test case is to verify that the actual Bluetooth implementation



Generated traceability matrix

responds correctly to multi-part responses. The ServiceSearchAttributeResponse message contains data that indicates to the client that the transaction continues and the client needs to enquire the server again. Therefore the last two messages in the test case notify the application level that the transaction was not complete; and then present a new inquiry to the server. Then the test case ends as at this point the goal of the test case (to verify the client's correct handling of a multi-part transaction) has been fulfilled.

The tool generates also traceability (see above) and test case dependency data (see below) and this data is also exported and can be processed outside the tool. For example, the requirements traceability data can be fed back to tool such as IBM RequisitePro or HP Quality



Generated test case dependency matrix

Center, and the test case dependency data can provided to the test management and test execution tools.

Test Execution

In order to execute the generated test cases, a test harness (connection to the actual system under test) is needed. Its nature depends on whether the system under test is an actual chip, simulated platform, or a software-only component of a Bluetooth implementation. The harness needs to be also accessible through a programming or scripting language. For example, if the implementation is a C++ software library, the test cases would be exported from Conformiq Designer in C++ and linked against the library. Or, if the implementation is an actual chip on a test platform, the test platform's drivers would be accessed through the platform's scripting language, to drive both the application interface as well as the air/radio interface.

Benefits of Computer-Generated Functional Tests

Letting computers to take care of the details of functional test design brings in multiple technical benefits:

- This algorithmic approach eliminates randomly incorrect tests.
- There are fewer missing tests, because the algorithm does not accidentally miss corner cases.
- There are fewer redundant tests because the resulting test sets are optimized by a computer and checked for importance.
- Test maintenance is easier because half of the job is automated.
- Test execution systems can be simplified, because they are no longer targeted towards human-written tests, but computer-written tests.
- Testing equipment costs decrease because the optimized test sets run faster on leaner test execution systems.
- Personnel cost can be optimized because the approach delivers a significant productivity improvement.
- Test design time shrinks because the algorithms do the design faster than humans.
- Traceability improves as it is now automatically maintained by a computer.
- Test documentation is always consistent and up-to-date as it is generated at the same time as the actual tests.

These technical benefits translate into business benefits that can impact a corporation's bottom line:

- 30–80% reduction in functional testing costs without sacrificing quality, as measured in our customer contexts
- Increased test coverage leads to fewer shipped defects which reduces the tax of escaped and customer-foudn defects on R&D and customer service
- Decreased time to market as the testing process can turn around faster
- Reduced risks of litigation, standards and process non-compliance, and loss of key knowledge when personnel changes

About Conformiq

Established first in 1998, Conformiq is a leading provider solutions for automated test design and advanced model-based testing, dedicated to improving test design processes within software-intensive product companies operating in business-, mission- and life-critical industry segments.

Conformiq Designer™ is the company's fourth-generation test design tool, built upon a decade of advanced basic and applied research as well as testing and test design experience.

Privately held, independent and known for extraordinarily responsive customer service, Conformiq is the partner of choice for companies who are ready to step ahead of the curve.

For more information about Conformiq and the company's offering, please visit www.conformiq.com.

Copyright © Conformiq Inc. and its subsidiaries 2010. All Rights Reserved. All information in this publication is provided for informational purposes only and is subject to change without notice. Conformiq, Conformiq Designer, Conformiq Modeler, Open Model Licensing and Automated Test Design are trademarks of Conformiq Inc. Other trademarks and registered trademarks belong to their respective owners.

```
public void sdpServiceSearchResponse(SDP_ServiceSearchResponse msg)
{
    try
    {
        msg.requireMe(servSearch.maxRecordCount);
        checkPDUBase(msg, 0x03);
        if (msg.paramLen >= 0 && msg.paramLen < 5)
        {
            requirement "SDP-client/SDP_ServiceSearchResponse/" +
                "Must not allow too small PDU parameter length";
            msg.requireEmpty();
            protoErrorParamLen();
        }
        if (msg.paramLen != (4 + 4 * msg.currentRecordCount +
                             msg.contState.byteSize()))
        {
            requirement "SDP-client/SDP_ServiceSearchResponse/" +
                "Must reject PDUs with non-matching parameter length";
            protoErrorParamLen();
        }
        if (msg.currentRecordCount > msg.totalRecordCount)
        {
            requirement "SDP-client/SDP_ServiceSearchResponse/" +
                "Must reject PDUs with invalid record counts";
            protoError("Current rec count cannot exceed total rec count");
        }
        if (msg.currentRecordCount * 4 != msg.recordHandleList.length)
        {
            requirement "SDP-client/SDP_ServiceSearchResponse/" +
                "Must reject PDUs with non-conforming PDU-content";
            protoError("Record count doesn't match the size of record " +
                       "handle array");
        }

        if (msg.recordHandleList.length != 0)
        {
            rspBuffer = concatByteArray(rspBuffer, msg.recordHandleList);
        }

        // OK, now we have the full buffer so far, check cont state
        // and continue if needed.
        if (msg.contState.infoLen == 0)
        {
            // Done with this transaction, return results!
            requirement "SDP-client/SDP_ServiceSearchResponse/" +
                "ServiceSearch results returned OK";
            ServiceResults results;
            results.recordHandles = rspBuffer;
            appOut.send(results);
            transactionDone();
        }
        else
        {
            // Not done, must continue!
            requirement "SDP-client/SDP_ServiceSearchResponse/" +
                "Continuation state was defined, must continue";
            TransactionContinued cont;
            appOut.send(cont);
            servSearch.tid = getNewTid();
            servSearch.contState = msg.contState;
            btOut.send(servSearch);
        }
    }
    catch (ProtocolException ex)
    {
        notifyProtocolError(ex.status, ex.msg);
    }
}
```