Actual Trends in Development of Power System Protection and Automation
7-10 September 2009, Moscow

A Tool for Testing and Fault Diagnosis in IEC 61850-Based Systems

Jacques Sauvé[1], Iony Patriota[2], Wagner Porto[1]
[1]Federal University of Campina Grande [2]Companhia Hidro-Elétrica do São Francisco
Brazil
jacques@dsc.ufcg.edu.br

KEYWORDS

IEC 61850 Substation Automation Systems, Functional Testing, Fault Diagnosis, Performance Testing.

1 INTRODUCTION

This paper describes a proof-of-concept software tool that enables automation engineers to build, run and debug functional tests for IEC 61850-based systems in a simulated environment. The paper is structured as follows. Section 1.1 described the work of Cigré WG B5.32 on Functional testing and section 1.2 describes the problems that this paper discusses. Section 2 describes an example of a functional test. Section 3 describes the solution to the problems in the form of a software tool used to build functional tests. Section 4 describes the architecture and prototype of the tool. Finally, section 5 concludes the discussion and provides ideas for future directions.

1.1 Context: Testing IEC 61850 systems

The introduction of the IEC 61850 has resulted high added value in the implementation of Substation Automation Systems (SAS). However, although conformance and interoperability tests are subject to standardized approaches, functional and performance testing are not yet subject to standards. Cigré Workgroup B5.32, entitled *Functional Testing of IEC 61850-Based Systems* was formed in 2006 to propose a solution for such testing activities.

The approach taken by WG B5.32 revolves around black-box testing which is a quality assurance process that verifies that an application's functionality works accurately, reliably, predictably and securely [1]. Functional testing consists of a series of tests that emulate the interaction between IEC 61850 intelligent electronic devices (IEDs) and the application in order to verify whether or not the application does what it was designed to do. The proposed solution allows the construction of "test scripts" that can verify functional behavior and performance characteristics.

The solution proposed by WG B5.32 is being submitted to IEC for standardization in the near future (as of this writing, in April 2009).

1.2 The problems

Two problems have been identified and are the reason for this paper:

- The work done by WG B5.32 needs a proof-of-concept implementation to test its viability in practice.
- A very useful extension to the testing process can be performed by analyzing the test results and producing a diagnosis of where faults may reside in an SAS.

The paper describes the architecture and preliminary results of a software solution to these problems called Smash.

2 FUNCTIONAL TESTING: AN EXAMPLE

Only a brief sketch of an example test scenario can be given here due to space restrictions. Please refer to the full WG B5.32 technical brochure for details [1]. The motivation for providing this example is to give the reader unfamiliar with WG B5.32's proposal an outline of the approach so that the rest of the paper may be better understood. The approach is object-oriented and UML, text and XML formats to specify the applications through Functional Use Cases and other Functional Specification documents. We do not show all these documents here due to lack of space.

Consider the substation layout diagram shown in Figure 1.

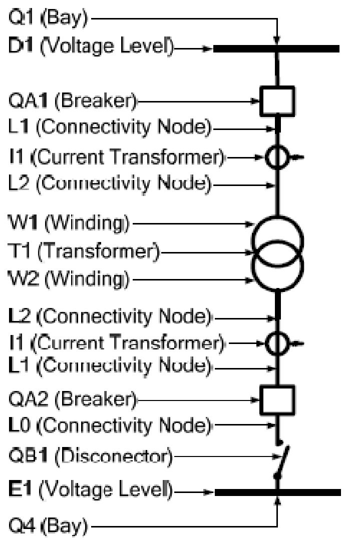


Figure 1: Example substation layout diagram

The functional specification of the system may include Functional Implementation Conformance Statements (not shown), specifying, for example, that XCBR1 and XCBR2 must trip in less than 100 ms upon inception of an internal short circuit in the transformer. In addition, other UML diagrams may be used in the functional specification. For example, a UML communication diagram is shown in Figure 2 and a UML sequence diagram is shown in Figure 3. In Figure 3, the numbers shown are PICOM types (12=Operated, 22 = Trip, etc.). The left-hand side of the figure also shows the performance requirements as time delay restrictions.

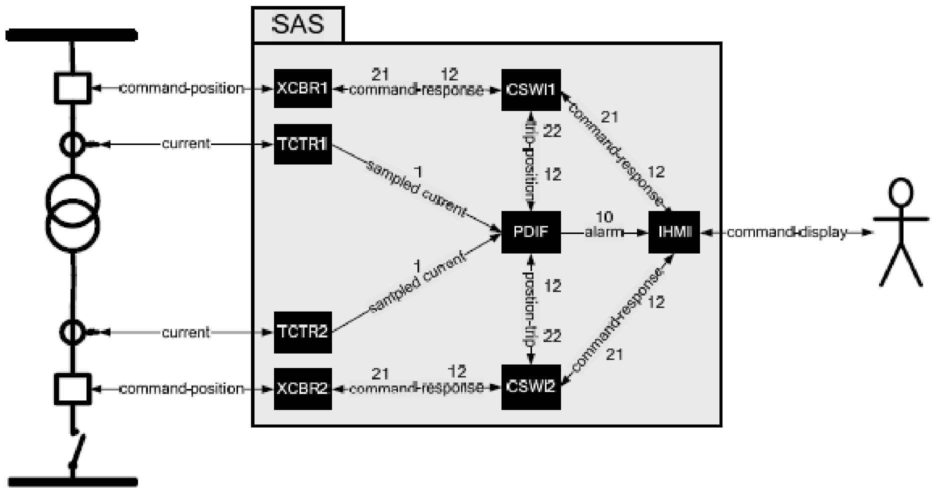


Figure 2: Functional specification by UML communication diagram

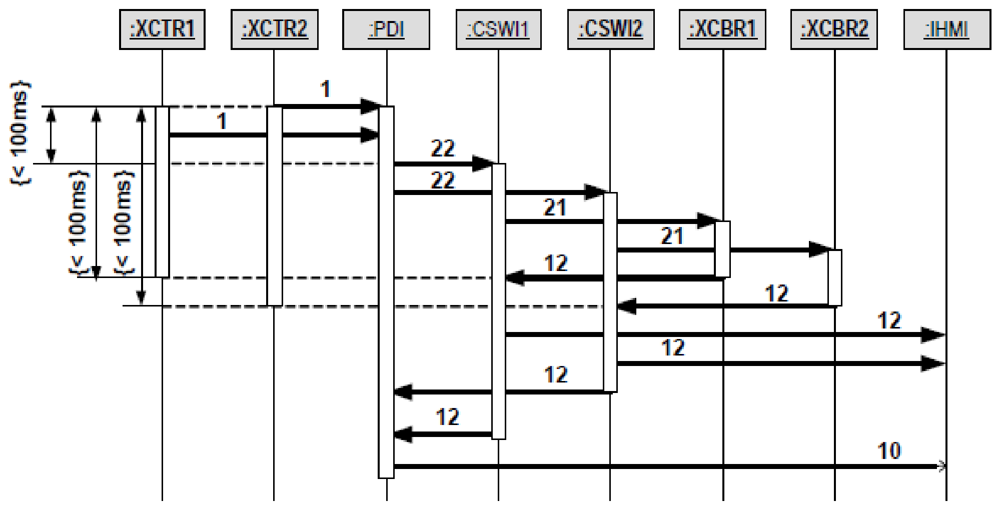


Figure 3: Functional specification by UML sequence diagram

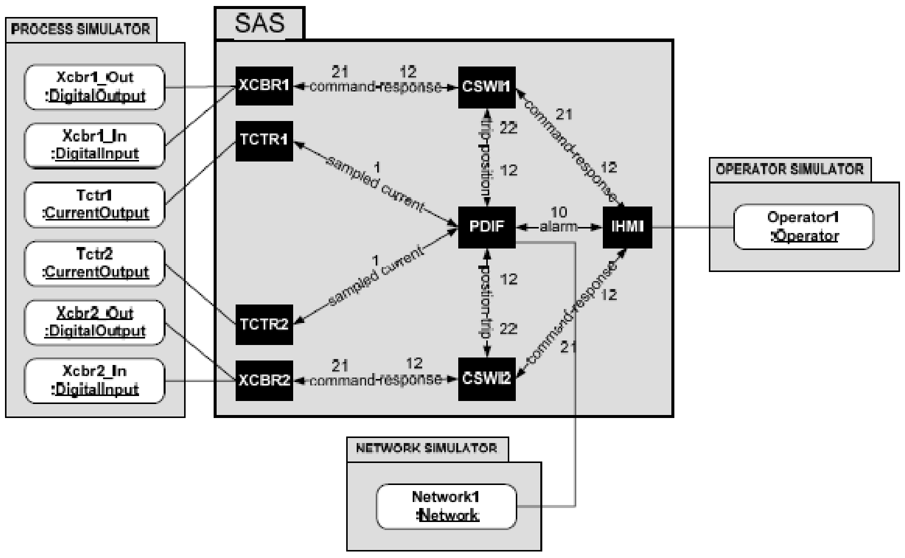


Figure 4: Test setup as a UML communication diagram

We now move on to the specification of functional tests. WG B5.32 recommends that Failure Modes and Effects Analysis (FMEA) and Hazard and Operability Analysis (HAZOP) be used tools to drive the design of tests and also to investigate the fault coverage attained by test plans. WG B5.32 has suggested a test architecture consisting of several test components used in automatic testing activities. Figure 4 shows the testing objects instantiated from the test device classes necessary to test this example SAS. The figure also shows their connection to the SAS Logical Nodes (LN).

Reference [1] describes this setup as follows: “Note that each breaker is modeled by a DigitalOutput and a DigitalInput object, to simulate their command and response messages, while each current transformer is modeled by a CurrentOutput object, to simulate their sampled currents. A network simulator (or analyzer) is instantiated and assigned to monitor the messages related to logical node PDIF, to measure its response time. Messages sent and/or received by the operator are modeled by an Operator object. This setup can be described more fully as a functional test case [. This is shown below] for the three functions specified in this example SAS.” As one can see, test scripts can specify signals to be injected in the system as well as signals to be expected.

	Test Connection	
1.1	Timer1 = TestTimer()	Create a timer to measure events
1.2	Arbiter1 = TestArbiter ()	Create a test arbiter to emit verdicts
1.3	Xcbr1_In = DigitalInput (XCBR1)	Create a digital input connected to XCBR1
1.4	Xcbr1_Out = DigitalOutput (XCBR1)	Create a digital output connected to XCBR1
1.5	Tctr1 = CurrentOutput (TCTR1)	Create an analog output connected to TCTR1
1.6	Tctr2 = CurrentOutput (TCTR2)	Create an analog output connected to TCTR2
1.7	Xcbr2_In = DigitalInput (XCBR2)	Create a digital input connected to XCBR2
1.8	Xcbr2_Out = DigitalOutput (XCBR2)	Create a digital output connected to XCBR2
1.9	Pdif = NetworkSimulator (PDIF)	Create a network simulator linked to PDIF
1.10	Operator1 = Operator (IHMI)	Create an operator connected to IHMI
	Test Setup	
2.1	Xcbr1_Out->SetDigitalOutput (1)	Prepare to close breaker XCBR1
2.2	Xcbr2_Out->SetDigitalOutput (1)	Prepare to close breaker XCBR2
2.3	Xswi_Out->SetDigitalOutput (1)	Prepare to close switch XSWI
2.4	Tctr1->SetACCurrentOutput (0,0)	Prepare to zero current on node TCTR1
2.5	Tctr2->SetACCurrentOutput (0,0)	Prepare to zero current on node TCTR2
2.6	Xcbr1_Out->StartDigitalOutput ()	Close breaker XCBR1
2.7	Xcbr2_Out->StartDigitalOutput ()	Close breaker XCBR2
2.8	Tctr1->StartCurrentOutput ()	Zero current on transformer TCTR1
2.9	Tctr2->StartCurrentOutput ()	Zero current on transformer TCTR2
2.10	Pdif->GetMessageSequence (1min)	Record messages for 1min to and from PDIF
2.11	Xcbr1_In->GetDigitalIinputSequence (1min)	Record input sequence for 1min from XCBR1
2.12	Xcbr2_In->GetDigitalIinputSequence (1min)	Record input sequence for 1min from XCBR2
	Test Start	
3.1	Tctr1->SetACCurrentOutput (5,0)	Prepare 5A on current on transformer TCTR1
3.2	Timer1->Start ()	Start time to measure function delays
3.3	Pdiff->StartNetworkSimulator()	Start recording messages to/from PDIFF
3.4	Time1=Tctr1->StartCurrentOutput ()	Apply 5A to node TCTR1 and record time
	Test Stop	
4.1	Wait (2min)	Wait for 2min without processing the script
4.2	Tctr1->SetACCurrentOutput (0)	Prepare to zero current on node TCTR1
4.3	Tctr1->StartCurrentOutput ()	Zero current on transformer TCTR1
4.4	Pdiff->StopNetworkSimulator()	Stop recording messages to/from PDIFF
	Test Disconnection	
5.1	Time2 = Pdif->FirstPICOMTo (CSWI1,22)	Get time of first trip from PDIF to CSWI1
5.2	Time3 = Pdif->FirstPICOMTo (CSWI2,22)	Get time of first trip from PDIF to CSWI1
5.3	Time4 = Xcbr1_In->FirstDownInputTransition ()	Get time of opening of breaker XCBR1
5.4	Time5 = Xcbr2_In->FirstDownInputTransition ()	Get time of opening of breaker XCBR2
	Test Verdict	
6.1	Verdict1 = Arbiter1->TestArbiterConfirm (Time2-Time1 <100)	Trip of PDIF to CSWI<100ms
6.2	Verdict2 = Arbiter->TestArbiterConfirm (Time3-Time1<100)	Trip of PDIF to CSW2<100ms
6.3	Verdict3 = Arbiter->TestArbiterConfirm (Time4-Time1<100)	Trip of breaker XCBR1<100ms
6.4	Verdict4 = Arbiter->TestArbiterConfirm (Time5-Time1<100)	Trip of breaker XCBR2<100ms
6.5	Verdict5 = Operator1->OperatorConfirm (“PDIF Trip”)	Confirm PDIF trip indication
6.6	Verdict6 = Operator1->OperatorConfirm (“XCBR1 Trip”)	Confirm XCBR1 trip indication
6.7	Verdict7 = Operator1->OperatorConfirm (“XCBR2 Trip”)	Confirm XCBR2 trip indication

Table 1: Example test script

Each command in this script is a method call supported by the instantiated class. The last 7 commands (verdicts) evaluate the results of the test case. These commands check the time performance of the SAS against the specification (<100ms), as well as operator notification of breaker trippings and operation of the differential protection. Test cases can also be specified in XML.

3 REQUIREMENTS FOR A TESTING AND DIAGNOSIS SOLUTION

The solution to the SAS testing and fault diagnosis problems outlined above must satisfy requirements which are now listed. The overall vision is to build a software tool that will allow automation and protection engineers to develop and debug SAS designs and check their correctness through test scripts. The requirements for the system are as follows:

1. In a first phase, the system will be used to build and debug tests in a simulated SAS environment only. In a second phase, the system may be used during actual operation on a real SAS, by injecting actual messages at appropriate SAS access points, recording appropriate messages and evaluating the performance and functionality through test scripts.
2. The system must be able to execute test scripts and report on the test verdicts. There must be full support for all B5.32 test objects (VoltageOutput, CurrentOutput, DigitalInput, DigitalOuptut, NetworkSimulator, Operator, TestTimer, TestScheduler, TestArbiter).
3. The SAS must be represented by a model and simulated during execution.
4. The LNs most commonly used in SAS design must be supported.
5. The design must be component-oriented to allow third parties to develop new LNs and plug them into the system.
6. The system must provide test script management (script creation, visualization, editing, removal)
7. The tool must read in SAS models provided in IEC 61850 Substation Configuration Language (SCL)
8. The tool must provide for visualization and editing of test scripts in a script language and also in XML.
9. The tool must provide automatic conversion between the script and XML versions of a test.
10. The tool must perform syntax checking during script editing.
11. The test execution environment must provide
    - Execution command: Run all, Run selected, Pause, Stop.
    - Debugging mode (Run debug, breakpoints, single step, variable watch)
    - Simulated time speed control to accelerate or decelerate the simulation as compared to real time.
12. The execution environments must provide mechanisms for the insertion of faults
13. The tool must provide fault diagnosis functionality through an automatic fault diagnosis algorithm, thus allowing the source of faults to be pinpointed, down to the level of Logical Node.

4 A SOLUTION: SMASH – SMART SAS TEST AND FAULT DIAGNOSIS

As tool called "Smash – Smart SAS Test and Fault Diagnosis" is being developed to satisfy the requirements outlined above. This section describes the tool's architecture, its interface and the current development status.

4.1 Smash Architecture

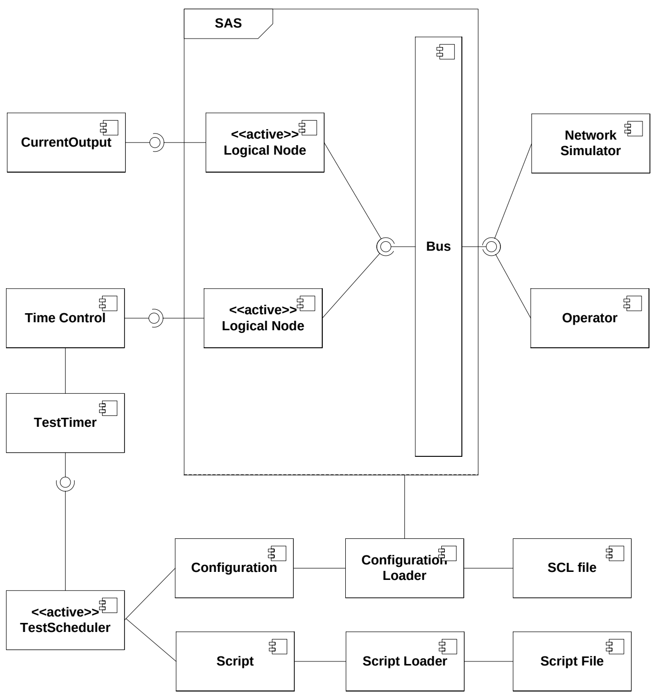


Figure 5: Smash Architecture

The Smash architecture is shown in Figure 5. The following are the main points to be appreciated in this figure.

- The SAS is represented by LNs which are components that simulate the behavior of functions such as differential protection (PDIF), circuit breakers (XCBR), etc., as per the 61850 standard. Since the architecture is componentized (the LNs are components that obey a standard discovery interface), new LNs can be added by third parties to the tool.
- The LNs are "active" classes, meaning that they run in a separate thread. This enables time delays to be introduced in their behaviors.
- All Publish-Subscribe communication between LNs and other test components is controlled by a common bus. This allows the simulated environment to include network delays in the simulation.
- The main data structures are the LNs themselves as well as the Configuration component containing an in-memory version of the SCL file and a Script component containing an in-memory version of the script being executed.
- The TestScheduler is the main simulator that interprets and executes script commands.
- Simulated time control is provided by the Time Control component. This is where speed control is implemented. All components requiring time service must interface with this component.

4.2 Smash User Interface

This subsection provides an outline of the Smash User Interface. Figure 6 is the main script execution screen. It consists of 5 main screen areas:

- At the top left is a menu that provides test script management, execution control etc.
- Immediately below the menu is a tool bar for test script execution and debugging , further detailed in Figure 7. Observe that a debugging mode is available to single-step execution, set breakpoints, examine the value of variables, etc.
- Below the tool bar is an area that shows the available tests. A test can be chosen and run from this list.
- Below, on the bottom left, is an area that more fully describes the functional test case and provides access to the "Functional Implementation Conformance Statement" (FICS) and "Functional Specification Requirement" (FSR) files and version information.
- Finally, on the write, the test script is exhibited, either in a scripting language (as shown in the figure) or as XML text.

Debugging allows one to set breakpoints. Figure 9 shows the script with two breakpoints with execution stopped at the first one. Finally, after execution, the results of all verdicts contained in the test can be examined (see Figure 8).

4.3 Current status and results

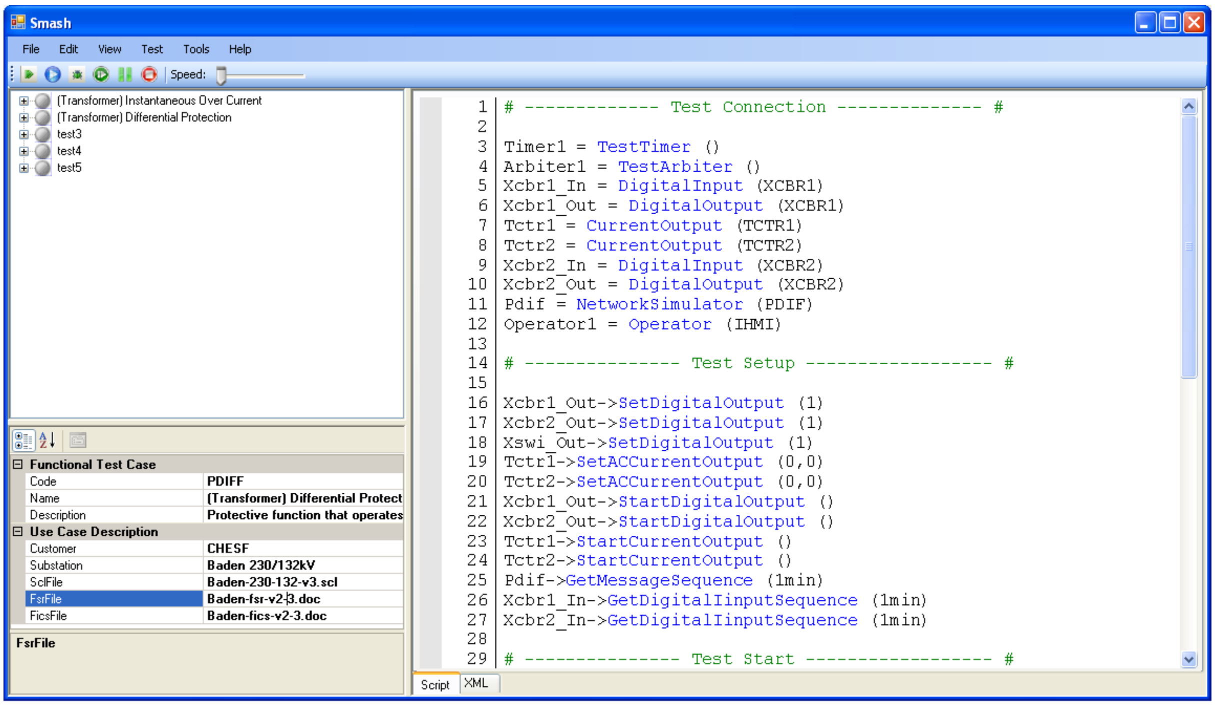
In April 2009, as of this writing, the tool described above is still being developed. Therefore, no results of actual use can be given at this point. However, we expect to have results, through actual use by engineers at CHESF in Brazil by the time of the conference in October 2009.

5 CONCLUSION AND FUTURE WORK

The problem we set out to solve is to provide a proof-of-concept implementation of a tool to help automation and protection engineers design and test SASs. The tool that was designed should be useful in two scenarios:

- Having a tool to simulate an SAS and apply tests to it will help automation and protection engineers to develop and debug SAS designs and check their correctness through test scripts.
- Once the test scripts are ready, they can be executed on a real SAS implementation to check that it meets functionality and performance requirements.

We believe that the design we are proposing will be useful in both scenarios, although we must still gather experience from real users to validate the idea fully.


Smash
File Edit View Test Tools Help
Speed:
(Transformer) Instantaneous Over Current
(Transformer) Differential Protection
test3
test4
test5
Script
XML

Functional Test Case	
Code	PDIFF
Name	(Transformer) Differential Protect
Description	Protective function that operates
Use Case Description	
Customer	CHESF
Substation	Baden 230/132kV
SclFile	Baden-230-132-v3.scl
FsrFile	Baden-fsr-v2-3.doc
FicsFile	Baden-fics-v2-3.doc

FsrFile

```
# ------------- Test Connection -------------- #

Timer1 = TestTimer ()
Arbiter1 = TestArbiter ()
Xcbr1_In = DigitalInput (XCBR1)
Xcbr1_Out = DigitalOutput (XCBR1)
Tctr1 = CurrentOutput (TCTR1)
Tctr2 = CurrentOutput (TCTR2)
Xcbr2_In = DigitalInput (XCBR2)
Xcbr2_Out = DigitalOutput (XCBR2)
Pdif = NetworkSimulator (PDIF)
Operator1 = Operator (IHMI)

# -------------- Test Setup ----------------- #

Xcbr1_Out->SetDigitalOutput (1)
Xcbr2_Out->SetDigitalOutput (1)
Xswi_Out->SetDigitalOutput (1)
Tctr1->SetACCurrentOutput (0,0)
Tctr2->SetACCurrentOutput (0,0)
Xcbr1_Out->StartDigitalOutput ()
Xcbr2_Out->StartDigitalOutput ()
Tctr1->StartCurrentOutput ()
Tctr2->StartCurrentOutput ()
Pdif->GetMessageSequence (1min)
Xcbr1_In->GetDigitalIinputSequence (1min)
Xcbr2_In->GetDigitalIinputSequence (1min)

# -------------- Test Start ----------------- #
```

Figure 6: Smash Main Screen

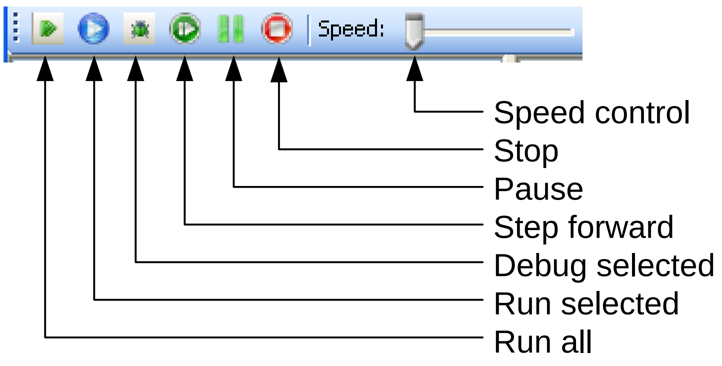


Figure 7: Test Script Execution Control Tool Bar

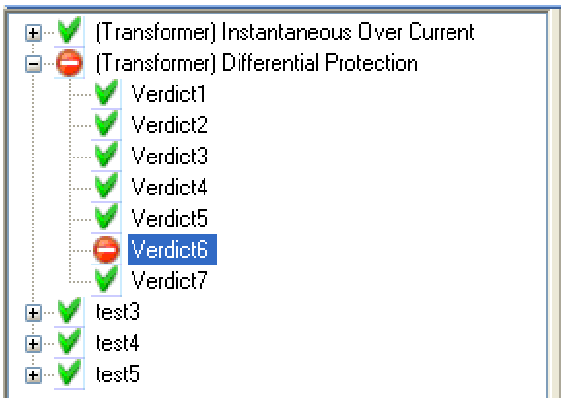
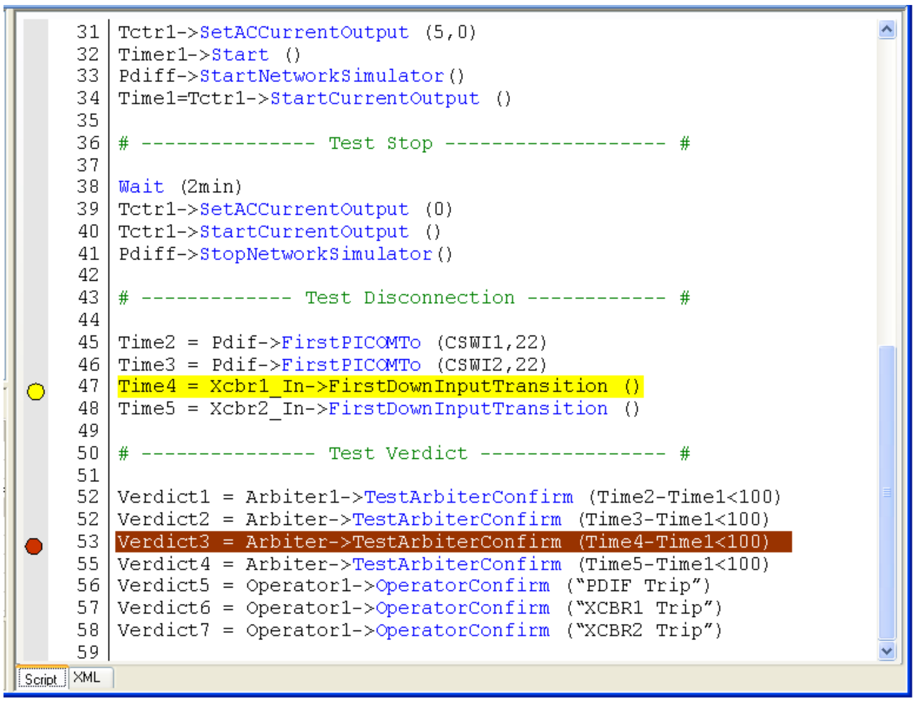

(Transformer) Instantaneous Over Current
(Transformer) Differential Protection
Verdict1
Verdict2
Verdict3
Verdict4
Verdict5
Verdict6
Verdict7
test3
test4
test5

Figure 8: Depicting Test Script Verdicts



```
Tctr1->SetACCurrentOutput (5,0)
Timer1->Start ()
Pdiff->StartNetworkSimulator()
Time1=Tctr1->StartCurrentOutput ()

# --------------- Test Stop ------------------- #

Wait (2min)
Tctr1->SetACCurrentOutput (0)
Tctr1->StartCurrentOutput ()
Pdiff->StopNetworkSimulator()

# ------------- Test Disconnection ------------ #

Time2 = Pdif->FirstPICOMTo (CSWI1,22)
Time3 = Pdif->FirstPICOMTo (CSWI2,22)
Time4 = Xcbr1_In->FirstDownInputTransition ()
Time5 = Xcbr2_In->FirstDownInputTransition ()

# --------------- Test Verdict ---------------- #

Verdict1 = Arbiter1->TestArbiterConfirm (Time2-Time1<100)
Verdict2 = Arbiter->TestArbiterConfirm (Time3-Time1<100)
Verdict3 = Arbiter->TestArbiterConfirm (Time4-Time1<100)
Verdict4 = Arbiter->TestArbiterConfirm (Time5-Time1<100)
Verdict5 = Operator1->OperatorConfirm (“PDIF Trip”)
Verdict6 = Operator1->OperatorConfirm (“XCBR1 Trip”)
Verdict7 = Operator1->OperatorConfirm (“XCBR2 Trip”)

```

Figure 9: Smash breakpoints

For the future, we envisage the following steps

- Validate the implementation with real users. This is planned for the second semester of 2009 at CHESF
- Implement more LN functions to enable varied SASs to be tested.
- Publish component interface specifications to enable third parties to build their own LNs and incorporate them in the tool.
- Design more complete test management functionality to manage the lifecycle of thousands of tests and test scenarios.
- Provide for the automatic generation of test scripts directly from SCL files and (perhaps) some additional performance objectives for the SAS.
- Provide for the automatic evaluation of coverage provided by a set of test scripts.
- Extend the tool to test real SASs by coupling Smash with “Protection relay injection test sets” and interfacing directly with the SAS communication network.
- Include fault diagnosis functionality. The input would be the verdict results from a set of tests and the output would be a set of suspected faulty IEDs.

We would like to thank the members of the Smart Diagnostics team at the Federal University of Campina Grande for the fruitful discussions and implementation effort.

REFERENCES

[1] *Functional Testing of IEC 61850-Based Systems Technical Brochure*, Cigré Workgroup B5.32, December 2008.