"Integrated Test Automation of IVR-Telephony Applications and Client-Server Call Center Applications"

Torsten Baumann; Test Team Leader,
          Interactive Media Group,
          Toronto, Ontario, Canada
Rex Black; President and Principal Consultant,
          RBCS, San Antonio, TX
Serban Teodorescu; Consultant,
          RBCS, San Antonio, TX
Gordon Page; Consultant
          RBCS, San Antonio, TX

Key Words : IVR, telephony, CTI, Client-Server, Test Automation,
          Test System Fake Pipe (TSFP), integration, simulated calls,
          Results, system under test.

Abstract

The Testing performed by our Test Group is Black Box Testing. This paper will discuss the following areas of our test effort in more detail:

- Client-server GUI testing
- CTI client-server testing
- IVR applications testing using simulated calls
- Product-wide integration testing using multi-tools (CallSim, Fake-Pipe, QA Partner) Approach
- Management considerations

Introduction

While test tools do exist for IVR Telephony, Client-Server and GUI Testing independently, there is not one that can test across the various systems. We have developed an approach that can be used in an integrated test environment.

The system under test (SUT) is described as follows. There are four main components:

- The Customer service application
- The IVR Telephony application
- The Content monitoring application
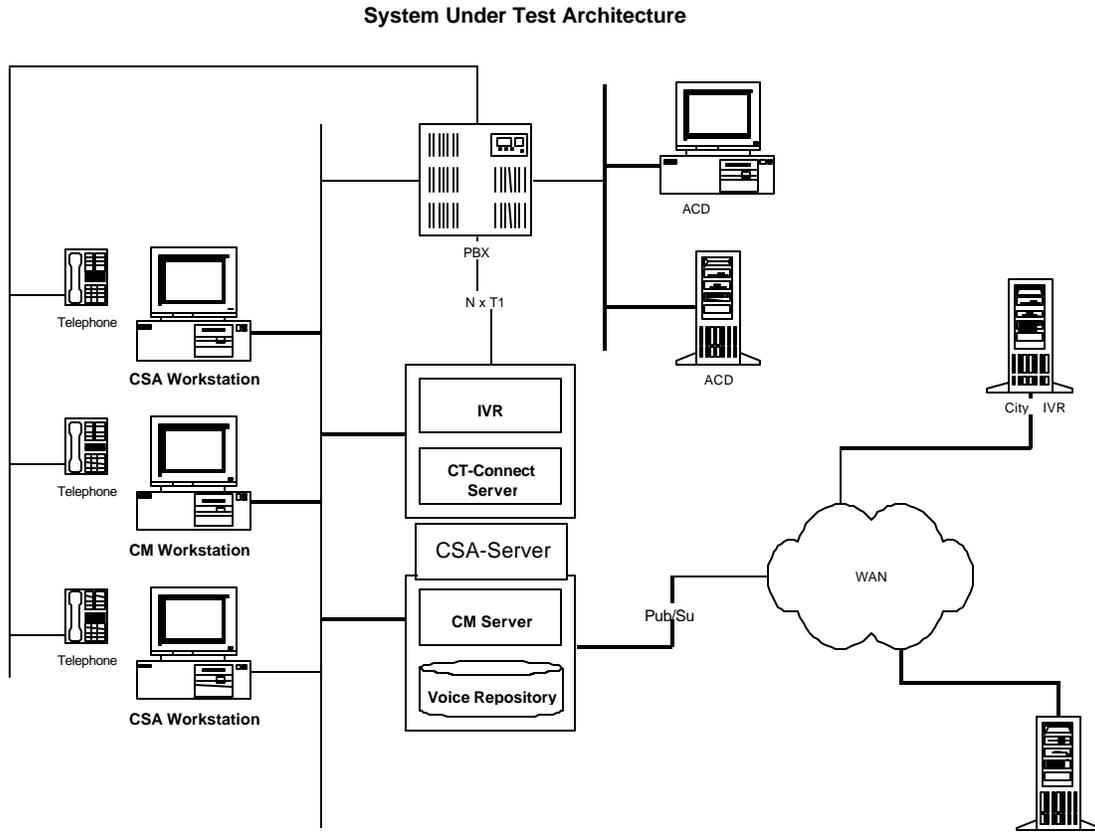- The CTI server and PBX.

[See figure 1.0]

System Under Test Architecture

<u>Figure 1.0</u>

The Client-Server GUI Testing will discuss the obstacles involved in developing and executing tests on a multi-form application developed in VB for applications.

The CTI Client-Server Testing will discuss the successes and difficulties involved in testing a CTI application developed in Visual C++.

The IVR applications module will allow the audience to understand the complexities involved in developing a call simulator in Perl as well as Tcl for stress, performance and path coverage testing.

The Product-wide Integration Testing using multi-tools approach will focus on the successes and issues associated with developing an integrated automation suite for this project. We will describe how test tools and programming languages widely available on the market can be combined by means of a "<u>Test System Fake Pipe" (TSFP)</u> to effectively test the entire data flow of the integrated system under realistic stress conditions. All failures are captured in a single easy to read report generated after test execution. All control points between the systems under test will be held in the TSFP file.

Last but not least the paper will discuss the **management considerations** involved in leading a test team made up of consultants, full-time test engineers and student test technicians in a project as large and as complex as this one.

Who we are Software Testing, QA or QC?

Quality Assurance:

These teams *assure quality*. You cannot do that by testing alone. True QA Staff is extremely senior otherwise there will be an issue of credibility. If this is the group that assures quality then the rest of the company does not assure quality. A true QA group must be involved at every stage of development; it must set standards, introduce review procedures, and educate people into better ways to design and develop products. It helps a company prevent defects. Here at iMG management holds this role.

This is not us.

Quality Control:

They do a lot of inspection and have the power ensure that a defective product not be released or be removed from production. We recommend to management that a defective product be pulled from the hopper. The QC Group will provide management with the following:
   a) Reports,
   b) Information gathering,
   c) Software testing of various products.

This group is a management assistant in that it informs management of product problems and the severity. Ultimately management decides.

This is what we are called.

Software Testers:

This is really what we are.

Project at a Glance

This project as all projects has had its ups and downs. The overall approach that was used for the entire project was as follows:
- Developers would Unit/Component and String Test their code.
- Developers would install the SUT in the test environment and Test would then perform a smoke test to determine if the system that was installed is in fact testable.
- If not, development would roll back the test environment to previous build. If testable, the testing cycle would begin.

Component, Integration, and System Test

When we first arrived at iMG, no formal testing by a test organization occurred prior to release to a test city for acceptance test. This strategy proved to be risky, since the deployment of an untested release candidate to a test city could result in some amount of downtime for that city's customer service staff due to software defects thus having a revenue impact for the company.

We then proceeded in implementing a test team, separate from the development team, to perform formal testing against release candidates prior to acceptance testing. This testing was generally divided into three overlapping phases:

Unit/Component Testing would basically cover the following areas:
- States, Transactions, Code Coverage, Functionality, Interface.

System Test would focus on each individual system with stubs around the integrated pieces so as to verify if the CT application would work standalone or the CSA application would work standalone.

This in no way showed that CSA would communicate effectively with CT application or vice-versa. The following quality risks are checked for:

- Functionality, User Interface, States, Transactions, Data Quality, Operations, Stress/Capacity, Load, Error Handling/Recovery, Installation, Standards, Configuration Options.

Integration Test would focus on having all major components of the entire project being put together, so to say, in the test environment which is a replicate of the production environment so that it can be tested from a black box approach. The quality risks verified for here were:

- Component Interfaces, Capacity and Volume, Error Handling and Recovery, Data Quality, Functionality, Usability, Performance.

In terms of release management during testing, the common practice that we used is the use the concept of "cycles". Each cycle takes place against a fresh build of the product, incorporating corrections that address all the "must-fix" bugs identified in the previous cycle.

A typical test phase, from the point of view of cycles, might follow the outline in the following table.

Test Phase	Cycle One	Cycle Two	Cycle Three	Cycle Four	Cycle Five
Component	Enter	Verify	Exit		
System		Enter	Verify-	Exit	
Integration			Enter	Verify	Exit

This table shows an idealized case. Each test phase goes through three cycles, with the entry criteria for the next phase met after the first cycle for the previous phase. The first cycle of each phase identifies some number of "must-fix" issues, but not enough to violate the entry criteria for the next phase. A verification build, with fixes, is made, and the test suite is rerun. At this point, perhaps only one or two issues remain for that suite, so a final build, with fixes, is made, and the suite rerun a third time, confirming the fixes and meeting the exit criteria for that phase.

Theoretically, testing may require only one cycle, if all three test suites fail to identify any "must-fix" issues. (One proceeds directly from component to integration to system testing without a new build, since no issues exist and no new build is required.) In practice, this seldom occurs.

We generally had three phases of test that included:

- Unit/Component and String test: This responsibility lay mainly with the development organization.
- System Test which the Test Team was accountable for. This was a six-week period with three two-week test cycles each.
- Integration Test which the Test Team was also accountable for. This was a six-week period with four cycles within.
-

Acceptance Test

When we arrived, acceptance test occurred by selecting a test city and deploying the latest release of the product there. After some length of time, the product, with any corrections arising from the acceptance test phase, exits acceptance test. The installation then proceeds.

This process proved to be costly, in that the Test City would occasionally receive builds that would cause great customer dissatisfaction thus impacting revenue in these test markets. This process was never formalized either. Some amount of "actual use" is difficult to reproduce in a test setting, so ad hoc testing by end users covers areas of the product that the test team will not. However, it is important that acceptance testing also follow a standardized test suite, including the following elements:

- Testing of the new functions, using the release notes and the requirement documents.
- Testing of critical operations; i.e., those functions that, were they not to work properly, could result in significant financial loss.
- Testing of typical operations; i.e., those functions most-frequently performed that, were they not to work properly, could result in inconvenience and inefficiency.

- Testing of recurring errors that surface from time to time; i.e., instability, occasional freezes under specific conditions, etc.

In addition, testing according to a standardized test suite entails allocating sufficient resources and time to complete execution. Therefore, acceptance test schedules should not derive solely from external priorities and deadlines.

At the end of the acceptance test, a "closed loop" process occurs against the requirement database. In other words, those requirements and issue reports that the end users acknowledge as resolved by the latest release should be marked as "closed".

Documentation

We developed and used the following five documents:

- **Maintenance Test Plan.** This overall plan addresses definitions, scope, risk management, failure mode effect analysis results, test configurations and environments, test execution process, test phases, entry and exit criteria for each phase, release management, issue tracking and analysis, status measurement, and test/development communication processes.
- **Component Test Suite.** This document will define a set of test cases to exercise individual functional components of the product.
- **Integration Test Suite.** This document will define a set of test cases to exercise the interfaces and connections between individual functional components of the product.
- **System Test Suite.** This document will define a set of test cases to exercise the overall behavior of the each separate component.
- **Version Status Report:** This document will allow one to view at a high level what was tested, what passed testing and what did not as well as the bug ids of any issues that have been opened against the system under test.

After the development of these documents, they undergo a peer review, including development and test staff. Then, the test team thoroughly tests the test suites themselves. The test team reviews each document during the specification and development phases for each release. Any enhancements or changes to the plan, processes or suites necessary to handle testing of the new release occur at that time.

Client Server GUI Testing

The Client server GUI Testing was one of the most successful projects in terms of early bug find/bug fixing. It was obvious from the outset that this project was one of the more organized in terms of design and clarity of specifications. This alone made the job of testing this application that much easier since it was clear to the test technicians what they were supposed to be testing and how. A prototype of the interface was also made available at an early stage so that the testers could start early with the automation in that the screen/window declarations and some of the primary functions could be created.

This application is created using Vantive, a third party supplier, but did necessitate many modifications to the Vantive code as we had many custom types of functionality required that Vantive simply did not offer. This application is running on Compaq workstations with Windows NT loaded and a minimum of application software and thus did not require extensive testing of different configurations. In past projects that Torsten Baumann has worked on he has found that testing for all possible configurations of workstations, was next to impossible since there was always someone that was running another application and thus system files may have been of different versions and the application would not work. This made our task that much easier as well, since all client desktops had the same file-set/image.

The server side resides on a Sun-UX box, which contains the Vantive application server, the Sybase database, Jaguar and Tibco.

The application under test was your basic Customer Service application with the regular features such as Member registration, Billing, Purchasing, Trial Offers, etc….

The Test Tool of choice for this application was QA Partner. The reason for this choice was quite simple actually. It was determined early in the project that this would be a GUI interface and a capture/playback tool would be used as it would be easier to train new testers with a small programming background in its use. Three of the Test Engineers that were hired on for this project also had experience using QA Partner in past projects and thus would not have to learn a new tool. Since Vantive is written with VBA and QA Partner can find most objects without much difficulty, we chose this approach. We generally would test the server by means of the client and QA Partner with all the piece parts running or stubs in place for those that were not yet ready for test.

Client Side Testing

There were been many discussion as to the approach we should be using. Should we do Client Side Testing or should we go right on the server and create tests there. In Keeping with the Black Box approach, the decision was made to go with Client Side Testing due to many different reasons some of which we will discuss below.

Traditionally one would think that creating tests from the client side will be easier simply for the reason that you may not need programmers as testers since the capture/playback did it all for you. This is not the reason we chose to go Client Side. We have found that what is generally known, as capture/playback in various test tools is not as simple as what some make it out to be. There is almost always a need for someone with some programming experience to make some modifications to actually make a useful test case. The main benefits we found were:

- The User Interface of the application will be tested at the same time.
- It was decided early in the project that the User interface will be frozen thus the test script maintenance would be kept to a minimum.
- This way we would be testing the application from the user perspective and results will be the same as those experienced by the users once the project went into production.
- We had the budget available for as many workstations as we needed to test that the system can handle certain loads. There was no concern that we would not have the hardware available to mimic the actual production environment.
- The middle-ware is exercised as well. Since Server Side Testing would not allow for the middle-ware to be tested under real-world conditions.

For this client server testing, testing scenarios followed a particular flow. The following incremental progression was used:

- Perform basic functional testing on a single client application so as to have those features needed for stress testing working effectively.
- Perform functional and concurrency testing on two remote client applications.
- Perform load testing of the server with a configuration large enough to stress the server (i.e. 100 users requirements)
- Perform extensive functional testing of all features.
- Perform regression testing.

Computer Telephony Interface Testing

The Configuration of the System under test is described as follows:

- CM server – NT server based
- CM client – NT workstation based
- CM IVR – UNIX based
- Servers application are C++
- Client application is Visual Basic

The CT System Operation is described as follows:

- IVR Member submits content via telephone
- IVR sends voice file and member information to CM server to access member profile
- CM server sends member profile and voice file information to CM-IVR
- CM IVR routes content to phone at CM client
- CM client displays member profile when call is answered
- CM client assigns status (approve/reject), call is disconnected, CM client is set to ready state
- CM server will also send status information to both Customer Service Application and the Home region IVR.
-

Manual Testing

Initial testing was done primarily in a manual fashion so as to gain familiarity with the application and to verify object content for those objects not recognized by the test tool. This involved using a utility (batch script calling C++ exe with DLLs), developed by the development team which was used to simulate incoming IVR telephone calls so that the Test Team could then simulate real-time calls into the CT system. This same DLL was used effectively to generate a background load on the application during Business Acceptance Test as well.

Manual Testing of this and all applications continues today, as one of our mandates is to ensure that the user will get the experience they crave when using the applications.

For this entire project (i.e. all subsystems) we have developed a test-case template that the engineers/testers will fill out in an easy to read sort of way and then at this point it was quite simple to have temporary workers and/or iMG employees execute the test-cases. The process was as follows:

- The testers (temporary worker, iMG employee) would be placed into teams of two (one playing IVR member and the other playing customer service representative).
- They would then be given a test-case, go back to a desk where a PC with the call center applications loaded, a telephone so they could dial the IVR as well as a second Lucent phone that was connected to the switch to route incoming calls from the IVR existed.
- The Tester would then begin doing each action item described in the test-case until the system under test would not function as described in the test case provided.
- At this point they would call a test technician who would validate the functionality, isolate the defect if existent and enter a bug report if required.
- The Testers would then go back and get another test case and so on.

See figure 2.0 for example of test case.

Action ID		Subsystem	Step	Person initiating the step	Status	Bug ID
	1	IVR	Dial in			
		IVR	Listen to Welcome message			
		IVR				
	2	IVR	Skip the changes			
	3	IVR	Press # at the login prompt			
	4	IVR	Choose Guest access			
	5	IVR	Choose Female			

6	IVR	Choose Intimate			
7	IVR	now at main menu			
8	IVR	Choose access the info center			
9	IVR	Choose exit			
9	IVR	Choose Record/Listen to success stories			
11	IVR	Choose record testimonial			
12	IVR	Record the following voice message "Testimonial, Female Guest, Intimate, Reject"			
13	IVR	Choose accept valid testimonial. This posts the testimonial to the CM			
14	CM	CM monitors and rejects the testimonial			

Figure 2.0

Automation development: Screen definitions

Those same utilities to simulate incoming calls that were discussed above were used to develop the automated test suite for this component. Some of the scenarios that we have lived through are:

- The Test Team was NOT part of the early design process. A problem encountered during this phase was that Visual Basic allows developers to not assign a handle to an object, this makes accessing objects for activities such as capturing text difficult if not impossible with automation tools. The lesson learnt by this is that a QA/QC or Test organization should be part of the design process, as retrofitting has impacted our schedules considerably due to this.
- If an object does not have a handle you may have to interact with it strictly by relative pixel location. Although this is not the optimal way to test it does work acceptably with objects such as buttons, check boxes, radio buttons, etc. Most tools will handle this acceptably well, if the GUI is sized so that the object is displayed.
- The CM client had several objects, buttons and text fields, which did not have handles; the buttons were interacted with by the above method. Text fields or boxes can not be; automation tools can not access text in an object they do not recognize. Unfortunately, one such text box had information related to the state of the GUI that was necessary to automate the testing. This required the developers to code additional utilities so that we can extract this information from other means besides the actual application.

Functional testing

- Because of the issues related to objects not having handles it was requested that the handles be added. Development felt that would negatively impact the schedule and possibly introduced more bugs. Their solution was to provide a window with the necessary status information when a command line parameter was set.
- The status window did provide the status information needed to automate the testing, but eventually we found it presented timing problems.
- Using utilities supplied by development we were able to simulate incoming content with profile information and telephone keypad key presses to test the CM client GUI functionality.
- The testing consisted of verifying the functionality of the various controls in the GUI and the proper update of the database.

Stress/Performance Testing

- Distributed processing – Host workstation connects to clients and sets up the client agent.
- A second workstation is set up to run a script that uses the DLL's to generate simulated calls. This script queues a specified number of calls per unit of time.
- When a call comes into a client the script running on the host communicates with the client to handle the call. The content was variously always approved, always rejected, or approved or rejected at the rate projected by marketing analysts.
- Initially, content was queued at a rate approaching development's projected maximum capacity. This led to problems, although the clients were handling content at an acceptable rate the servers quickly hung due to an inability to handle the load.
- Content queuing rates were lowered until the servers were able to handle the load. Then we slowly started increasing the rate until the problems started occurring again; this allowed development to find the problem.

Product Wide Integration Testing Using Multi-tools

Create a Test Software and Hardware Infrastructure That Allows Automated Testing of a Heterogeneous Enterprise Wide System

Tools Used:

- QA Partner for CT and CSA interfaces
- Perl, Tcl, Unix Shell Scripts for IVR Telephony interfaces
- TSFP (Test System Fake Pipe) (see figure 3.0)

Problems to overcome:

- The subsystems involved in the project run each on different platform using different operating systems. Each subsystem has a different kind of interface, based o a different set of paradigms, and a different type of test tools for test purposes.
- Testing must be non-intrusive; therefore the target applications are running in production mode (no debug and/or test configurations) as much as possible.
- The subsystems under test interact widely and continuously; therefore we must be able to test the system as a whole.

Solutions

- We must find a way to make the test drivers for each subsystem talk to each other outside the (sub) system(s) under test.
- The test feedback loop must be able to follow all the data transformations as the information flows across the system under test.

Implementation

- For each test thread, we establish a pool of data containing information about the state of each test driver involved as well as the associated test data. Each test driver is aware of the states of all the other test drivers involved, and is responsible to make correct test decisions based on this information.
- The integrated test applications use a state driven architecture:
  - The first test driver launches the tests and prepares the state information pool. It then drives its target subsystem through the programmed test flow, and posts the associated state transition info and test data to the state information pool. It pauses and waits for the next test driver to execute.
  - The second test driver was monitoring the state information pool. It senses the state transitions, drives its target subsystem through the programmed test flow, and posts the associated state transition info and test data to the state information pool. It then waits for the next test driver to execute.

- For the purposes of this presentation, we assume there are only two subsystems under test involved. The first test driver senses the state transition, wakes up and drives its subsystem into a "verification of the results" operation. It posts the info and exits.
- The second test driver senses the state transition, wakes up, and executes a similar operation for its target subsystem and exits.

Other Considerations:

- Each integrated test thread is independent. Any number of them, subject to resource availability, can execute simultaneously, enabling us to execute realistic test scenarios for the system under test.
- The implementation is very flexible. We use a .ini file structure with a general section, and a particular section for each subsystem (and its test driver) under test. Any type of test tool that can access text files over a TCP/IP network can be part of this test harness.
- The implementation is highly scalable. We can add any number of subsystems to our integrated test infrastructure by extending the state info pool with a corresponding section.

Test System Fake Pipe

The Test System Fake Pipe (TSFP) is used so that the telephony test driver and the client server test drivers can communicate with one another. Below you will find some sample code that locks, reads from and updates the TSFP as well as an example TSFP .ini file follows.

QA Partner sample code:

```
[-] type TsfpCMData is record //read from *.ini
        [ ] string sGender
        [ ] string sProduct
        [ ] string sRegion
 [+] type TsfpCMResults is record//write into *.ini
        [ ] string sAdSuccess
        [ ] string sGrSuccess
        [ ] string sIVRState
        [ ] string sCSAState
        [ ] string sCMState
        [ ] string sStartTime
        [ ] string sFinishTime
[+] // SetUpCMActions () comments
        [ ]
//*********************************************************************************
*
        [ ] //
        [ ] // SetUpCMActions ()
        [ ] //
        [ ] // Purpose:
        [ ] //    Reads the fake pipe .ini, and creates action file in QAP host workstation
        [ ] //
        [ ] // Usage:
        [ ] //   SetUpCMActions (sTSFPFile, sTestPath)
        [ ] //
        [ ] // Parameters:
        [ ] //   sTSFPFile - lstring; contains name of .ini file to read for test parameters
        [ ] //   sTestPath - string; path where the actions file is to be written
        [ ] //
        [ ]
//*********************************************************************************
*
```

```
[+] TsfpCMData SetUpCMActions (string sUAN, string sTSFPPath)
        [ ] string sINFileName
        [ ] FILEINFO fFile
        [ ] list of FILEINFO lfInfo
        [ ] IFILE ifTSFPFile = IFile ()
        [ ] TsfpCMData tParms
        [ ] string sPattern
        [ ]
        [ ] sPattern = "*{sUAN}*.*"
        [ ] PRINT (sPattern)
        [-] // do
                [ ] lfInfo = SYS_GetDirContents (sTSFPPath)
                [ ] Print (sTSFPPath)
                [-] Print (lfInfo)
                        [-] for each fFile in lfInfo
                                [-] if ( !fFile.bIsDir )
                                        [-] if ( MatchStr (sPattern, fFile.sName) )
                                                [ ] ifTSFPFile.Open (sTSFPPath,fFile.sName)
                                                [ ] // Get test parameters from the .ini file
                                                [ ] tParms = GetTsfpCMData (ifTSFPFile)
                                                [ ] Print (tParms)
                                                [ ] return tParms
        [-] // except
                [ ] LogError("File for UAN: {sUAN} not found")
        [+] return tParms
                [ ]
[-] // GetTsfpCMData () comments
        [ ]
//**************************************************************************************
*
        [ ] //
        [ ] // GetTsfpCMData ()
        [ ] //
        [ ] // Purpose:
        [ ] //   Reads a fake pipe .ini, gets the CM parameters from that .ini file and returns that
        [ ] //   information
        [ ] //
        [ ] // Usage:
        [ ] //   GetTsfpCMData (ifTSFPFile)
        [ ] //
        [ ] // Parameters:
        [ ] //   ifTSFPFile - IFILE; name of .ini file to read for a test's parameters
        [ ] //
        [ ]
//**************************************************************************************
*
[-] TsfpCMData GetTsfpCMData (IFILE ifTSFPFile)
        [ ] //
        [ ] //Section Headers
        [ ] string sRegistration = "Registration"
        [ ] string sUpdateMemberInfo = "UpdateMemberInfo"
        [ ] string sAds = "Ads"
        [ ] string sGreetings = "Greetings"
        [ ] string sState = "State"
        [ ] string sData = "CMTime"
        [ ] //
```

```
        [ ] //Values
        [ ] string sGender = "Gender"
        [ ] string sProduct = "Product"
        [ ] string sRegion = "Region"
        [ ] //Test
        [ ] string sStartTime = "StartTime"
        [ ] string sFinishTime = "FinishTime"
        [ ] TsfpCMData tParms
        [ ]
        [ ] tParms.sGender = ifTSFPFile.GetValue(sData,sGender)
        [ ] tParms.sProduct = ifTSFPFile.GetValue(sRegistration,sProduct)
        [ ] tParms.sRegion = ifTSFPFile.GetValue(sRegistration,sRegion)
        [ ] return tParms
        [ ]
[-] // SetTsfpCMData () comments
        [ ]
//*********************************************************************************
*
        [ ] //
        [ ] // SetTsfpCMData ()
        [ ] //
        [ ] // Purpose:
        [ ] //   Reads a CM results file and then updates the  .ini file
        [ ] //
        [ ] // Usage:
        [ ] //   SetTsfpCMData (ifTSFPFile, sTestPath, sUAN)
        [ ] //
        [ ] // Parameters:
        [ ] //   ifTSFPFile - IFILE; name of .ini file to update with a test's results
        [ ] //   sTestPath  - string; path where CM results files are located
        [ ] //   sUAN - string; content ID for a test, used to generate CM results file name
        [ ] //
        [ ]
//*************************************************************************** *********
*
[-] boolean SetTsfpCMData (IFILE ifTSFPFile, TsfpCMResults tCMResults)
        [ ] //
        [ ] //Section Headers
        [ ] string sAds = "Ads"
        [ ] string sGreetings = "Greetings"
        [ ] string sState = "State"
        [ ] //
        [ ] //Values
        [ ] string sStartTime = "StartTime"
        [ ] string sFinishTime = "FinishTime"
        [ ] string sAdSuccess = "CMSuccess"
        [ ] string sGrSuccess = "CMSuccess"
        [ ] string sIVRState = "IVR_State"
        [ ] string sCSAState = "CSA_State"
        [ ] string sCMState = "CM_State"
        [ ] //
        [ ] //Test
        [ ] string sSection
        [ ] boolean bDone = False
        [ ]
        [-] if tCMResults.sAdSuccess != NULL
```

```
            [ ] ifTSFPFile.SetValue (sAds, sAdSuccess, tCMResults.sAdSuccess)
            [ ] ifTSFPFile.SetValue ("CMTimeAD", sStartTime, tCMResults.sStartTime)
            [ ] ifTSFPFile.SetValue ("CMTimeAD", sFinishTime, tCMResults.sFinishTime)
     [-] if tCMResults.sGrSuccess != NULL
            [ ] ifTSFPFile.SetValue (sGreetings, sGrSuccess, tCMResults.sGrSuccess)
            [ ] ifTSFPFile.SetValue ("CMTimeGR", sStartTime, tCMResults.sStartTime)
            [ ] ifTSFPFile.SetValue ("CMTimeGR", sFinishTime, tCMResults.sFinishTime)
     [ ] ifTSFPFile.SetValue (sState, sIVRState, tCMResults.sIVRState)
     [ ] ifTSFPFile.SetValue (sState, sCSAState, tCMResults.sCSAState)
     [ ] ifTSFPFile.SetValue (sState, sCMState, tCMResults.sCMState)
     [ ]
     [ ]
     [-] while ( !bDone )
            [-] if ( ifTSFPFile.SetLock ("CM") )
                    [ ] ifTSFPFile.Close ()
                    [ ] ifTSFPFile.iStatus = DONE
                    [ ] ifTSFPFile.ClearLock ("CM")
                    [ ] bDone = TRUE
                    [ ] print ("result = {tCMResults.sStartTime} - (clear lock)")
            [+] else
                    [ ] sleep (2)
     [ ]
     [ ] return bDone
[+] boolean GotResults (string sTestPath, string sFileName)
     [ ] boolean bGotResultsFile = FALSE
     [ ]
     [+] if ( hHost -> SYS_FileExists ("{sTestPath}\{sFileName}.out") )
            [ ] bGotResultsFile = TRUE
     [ ] return bGotResultsFile
[ ]
```

Perl Sample Code:

A wrapper module exists that we call StateDriver written in Perl which is used for fakepipe interaction. It's usage by a script as follows:

```perl
# Tells script to use statedriver module
use StateDriver;

# loads the current tsfp_filename into an array
$tsfpFiles[$index] = $tsfp_filename ;

# Sets the TSFP filename in the statedriver module for all subsequent calls
&StateDriver::SetTSFPFileName(@tsfpFiles) ;

# Asks statedrive for UAN field under header Data (index is the index in the array and 60 is the # timeout)
my ($uan)= &StateDriver::GetKey("Data", "UAN", $index, 60) ;

# Tells Statedriver to change State information to IVR_State=GO, CSA_State=WAIT and #
CM_State=FINISHED
&StateDriver::SetState("GO", "WAIT", "FINISHED", $index, 60);

# Tells Statedriver to make PreTestCSABalance field to 50 under header Data
&StateDriver::UpdateKey("Data", "PreTestCSABalance","50", $index, 60);
```

These calls make TSFP interaction easier and allows code to be cleaner as TSFP calls can become lengthy and make code look messy. Two of the module procedures follow.

```
# 2 procedures from StateDriver.pm

sub StateDriver::GetKey {

  my($section) = shift(@_) ;
  my($keyValue) = shift(@_) ;
  my($ini_selection) = shift(@_) ;
  my($time_out) = shift(@_) ;

  my($count) = 0 ;
  my($tsfp_file) = $tstTSFP_File[$ini_selection] ;

  my($tsfp_fp, $tsfp_head) ;

  until(($tsfp_fp = &tsfp::tsfp_get_lock($tsfp_file)) ||
      ($count++ ge $time_out) ) {
    print " Waiting On Fakepipe .....","\n" ;
  } ;

   $tsfp_head = &tsfp::tsfp_load($tsfp_fp) or ( (&tsfp::tsfp_clear_lock($tsfp_head, $tsfp_file, $tsfp_fp))
&&
                    ( return ) ) ;
  my($valueOnTsfp) = &tsfp::tsfp_retreive_key($tsfp_head, $section , $keyValue ) ;
  &tsfp::tsfp_write($tsfp_head, $tsfp_fp) ;
  &tsfp::tsfp_clear_lock($tsfp_head, $tsfp_file, $tsfp_fp) ;
  return $valueOnTsfp ;
}


sub StateDriver::UpdateKey {

  my($section) = shift(@_) ;
  my($keyValue) = shift(@_) ;
  my($tstResult) = shift(@_) ;
  my($ini_selection) = shift(@_) ;
  my($time_out) = shift(@_) ;

  my($count) = 0 ;
  my($tsfp_file) = $tstTSFP_File[$ini_selection] ;

  my($tsfp_fp, $tsfp_head) ;

  until(($tsfp_fp = &tsfp::tsfp_get_lock($tsfp_file)) ||
      ($count++ ge $time_out) ) {
    print " Waiting On Fakepipe .....","\n" ;
  } ;

   $tsfp_head = &tsfp::tsfp_load($tsfp_fp) or ( (&tsfp::tsfp_clear_lock($tsfp_head, $tsfp_file, $tsfp_fp))
&&
                    ( return ) ) ;
  print STDERR "$section\t$keyValue\t$tstResult", "\n" ;
  &tsfp::tsfp_update_key($tsfp_head, $section , $keyValue, $tstResult) ;
  &tsfp::tsfp_write($tsfp_head, $tsfp_fp) ;
```

```
&tsfp::tsfp_clear_lock($tsfp_head, $tsfp_file, $tsfp_fp) ;
 1 ;
}
```

Figure 3.0: **TSFP.ini template**
[Registration]
RegistrationMethod=
Product=
Region=
VoiceApproval=
ServiceAgreement=
UANGeneration=
VoicePrintWaitTimeout=
CMVoiceApprovalSuccess=
IVRSuccess=
CSASuccess=
CSAStartTime=
IVRStartTime=
CMStartTime=
CSAEndTime=
IVREndTime=
CMEndTime=
CMWarn=
CSAWarn=
IVRWarn=
IVRError=
CSAError=
CMError=

[UpdateMemberInfo]
Method=
ChangeUAN=
ChangePPC=
ChangeZIP=
ChangeUANPPC=
NewUAN=
NewPPC=
NewZip=
IVRSuccess=
CSASuccess=

[Content]
Segment=
ExpectedResult=

Timeout=
CMSuccess=
CSASuccess=
IVRSuccess=
CSAStartTime=
IVRStartTime=
CMStartTime=
CMEndTime=
CSAEndTime=
IVREndTime=

[State]
IVR_State=
CSA_State=
CM_State=

[Data]
Gender=
PostalZip=
BirthDate=
MemberUAN=
UAN=
PPC=
PreTestIVRBalance=
PreTestCSABalance=

Management Considerations

In addition to the technical challenges facing the IMG Test Team, a number of complex management considerations arose. Many of these arose in three main areas: logistics, staffing the team, and tracking bugs and test cases. The following paragraphs discuss some of these challenges and how we managed them.

Logistics

The IMG facilities are located in two areas of the Greater Toronto Metropolitan Area. The headquarters are on King Street West in downtown Toronto, just blocks from the financial district. The call center, though, is located in Etobicoke near the Lester Pearson Airport. These two locations are about fifteen kilometers apart, which is usually fifteen minutes' drive during non-peak traffic hours. Because of the hardware requirements of the testing (see below), some testing took place in each location. This lead to a distributed testing approach. '
Having two test teams separated by only fifteen kilometers may seem trivial. It would appear less challenging than some test efforts one of the authors, Rex Black, has managed, which spanned the Pacific Ocean and involved participants in Taiwan, Japan, Northern California, Southern California, and Utah. However, in this case, the expected results of a test in one location depended heavily on the testing and status in the other. For example, if an IVR server went down on King Street, this could affect what the testers were seeing at Etobicoke. Likewise, CSA server crashes could delay and even lose data sent to the IVR server via the Send/Listen mechanisms provided by TIBCO.
The information transfer challenges created by this tight coupling of components were made even more difficult by the fact that we performed both manual and automated testing. The manual testing involved large teams who had to be able to start and stop within moments. The automated tests, likewise, needed

precision control based on an accurate picture of the current status of all the hardware, software, and network components. Single points of failure and latency made the status even less clear-cut.

The complex hardware, software, and network test environment simulated an even more complex hardware, software, and network field environment. We had a pair of IVR servers acting as deployed systems with two more IVR servers driving them with automated tests downtown, and a handful of various servers and telephony components, along with almost 100 client systems, at the call center. The IVR servers ran Unix, as did some of the telephony and database components. However, other servers and the client desktop systems ran Windows. Custom, customized, and COTS software ran throughout the environment. Finally, the network itself was a custom amalgamation of Ethernet, ISDN, and WAN elements. At least we only had a couple IVR servers located in the same town as the call center. The deployed system would have dozens of IVR servers, located singly and in pairs throughout North America, Australia, and, ultimately, Europe.

Information transfer challenges were dealt with three ways. First, the test team managers and the engineers spent a fair amount of time at each site, regardless of their direct responsibilities. This lead to rapid resolution of communication breakdowns and sympathy for each other's problems. Second, cell phones were issued to key players, individual contributors and managers, to make sure that escalation of problems and communication hurdles could happen instantly. Finally, shared bug and test tracking tools allowed each team to see the concrete results of the other's testing, even between detailed test status review meetings. The network complexity issues were dealt with by careful monitoring of status and instant escalation to outside system administration support when problems arose.

Test Team Size and Composition

The team was composed of a mixture of IMG full-time employees, IMG part-time employees, and RBCS consultants. About fifty people participated in the test effort at one point, though the core team was comprised of about fifteen people. The RBCS consultants between them have over 50 years of test experience, while the IMG employees tended to be new to software development projects or testing specifically. Therefore, the consultants served as both implementers and transferors of knowledge.

The exception to the IMG test team experience levels was the permanent IMG test manager, Torsten Baumann, who has a degree, hands-on management experience, and software testing experience. He brought his familiarity with test automation and relational databases to bear in a technical capacity, too, designing and implementing the Customer Service Application and Customer Data Repository tests, both manual and automated. Rex Black, the RBCS consultant test manager, got the process going initially, developing a budget, schedule, and plan for System and Integration Test. Once in Integration Test, he served primarily to assist Torsten in his efforts.

In addition to the test management professionals, a total of five test engineers worked on the project. One, Barton Layne, worked specifically on designing and implementing IVR testing. Another, Gordon Page, worked primarily on Content Management testing, especially automation of these tests and "marrying" that automation—as described in this paper—to the automation on the IVR side. Yet another test engineer, Serban Teodorescu, worked on the integration testing, addressing not only the intercommunicating pieces but also the testing of the "glue"-the Publish/Subscribe mechanism-that allowed the components to talk. All three of these engineers are RBCS consultants. Two more test engineers-at the time students at the University of Toronto-worked creating test tools such as the telephony load and traffic generator, CallSim, and the CallSim Test System Fake Pipe library.

Supporting both automated and manual testing were eight test technicians, a mixture of students and full-time professionals. At the peak of the manual testing, about a dozen temporary workers were brought in from a local staffing agency to run tests over a sixteen-hour-per-day period. Also, about two or three dozen IMG employees augmented the testing. Some of the Customer Service Representatives performed their jobs but with simulated IVR users who were actually testers. Other IMG employees simulated IVR use to generate load and to surface interface usability problems.
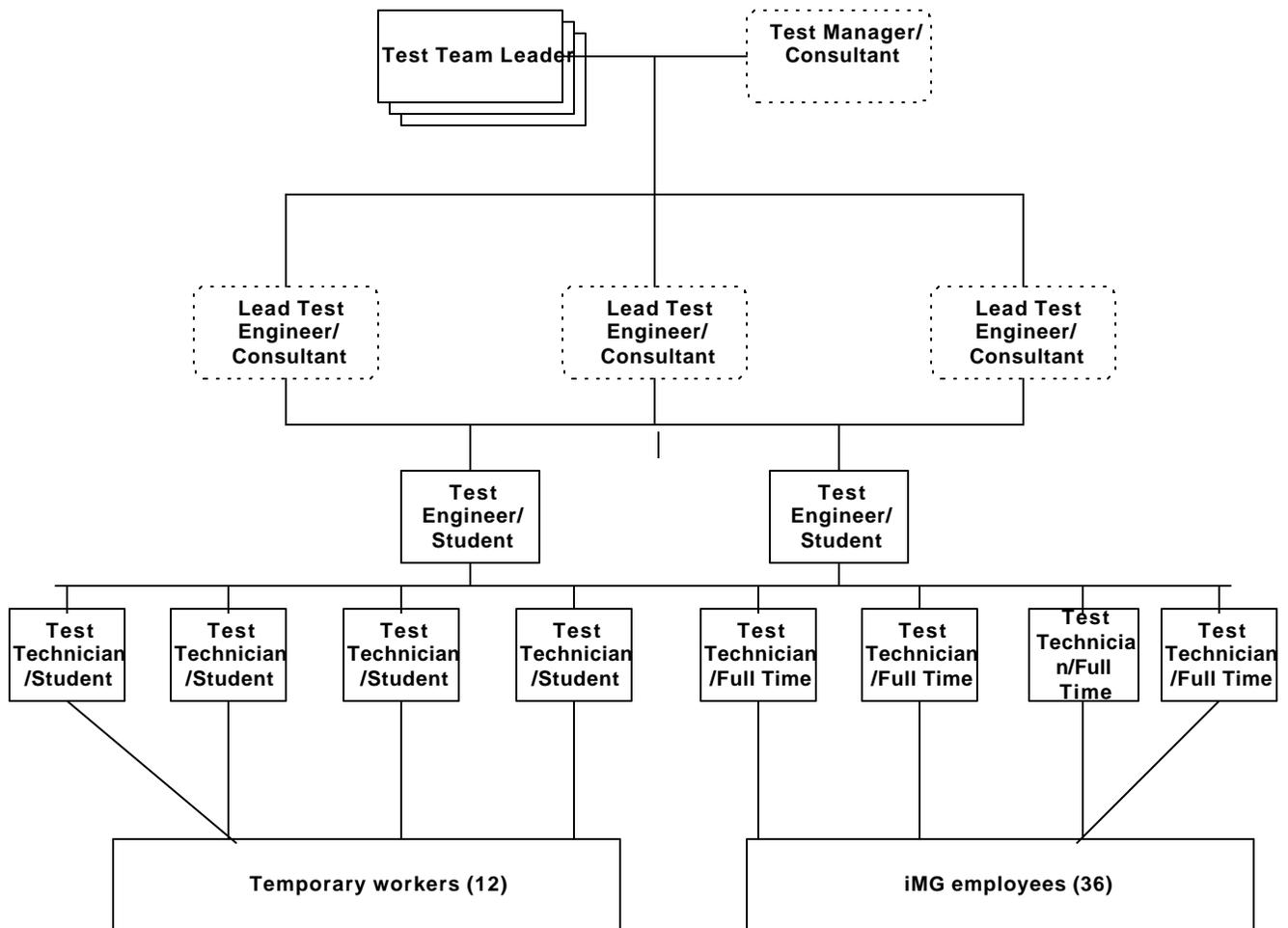
Such a test team may strike some as eclectic, but the approach is an unfortunately underutilized but excellent way to bootstrap a test organization into existence. Getting an independent test team off the ground is tough, and around 25 percent of such groups are disestablished within two years of their creation. It's important to build a solid foundation for future growth and to produce visible results immediately. By using seasoned professional test consultants, the test team was able to avoid many dangerous pitfalls, perform valuable testing right away, and design a solid test system architecture for the team going forward. Once the initial hurdles were passed, the consultants phased out, leaving behind the experienced test manager and a team of less experienced but now battle-proven testers who had now seen the job done properly and had inherited a professional-quality test system.

We did suffer from some drawbacks related to the non-test staff participation, namely trouble with writing decent bug reports. The test engineers and technicians used a review process to improve bug reports, but they had a common set of expectations about what constituted a "good" bug report, including detailed steps to reproduce and proper isolation. Amateur testers had no idea how to communicate problems effectively to development. In cases where the test team supervised and reviewed the amateurs' test results, considerable time was spent getting the reports in a usable format. In cases where the test team did not screen bugs reports, the reports were generally useless to development.

Bug and Test Tracking

An important challenge facing any test organization is the orderly tracking and management of the tests performed the bugs found, and the risks to quality addressed and abated by these results. Various commercial-off-the-shelf-software tools exist to perform these roles, but, since none were perfect fits for the complex, heterogeneous environment discussed in this paper, IMG decided to license a customizable solution from RBCS. The three components of this solution provide an integrated, quantitative test management system for bugs, tests, and risks to quality. As part of the test effort, the test team adapted these tools to handle these essential tasks.

The foundation of any well-engineered test system is a risk-based approach to test development. Which problems, if they occur, will seriously harm the users' experiences of quality, and which other bugs constitute mere nuisances? A quantitative approach to answer this question was provided by the Failure Mode and Effect Analysis technique.[1] The toolkit extended this technique by measuring test case coverage against the risks to quality identified through FMEA, and, by applying numerical ratings to coverage, giving management a yardstick for benchmarking quality testing.

As tests are developed, one must have a standard approach to documenting them. Once execution starts, one must track the results and status of each test case, and be able to report summary information on all the test suites across each test effort. A three-tiered method, based on Excel worksheets, provided information at the test step, test case, and test suite level. The lowest-level documentation provided the exact steps required for the test technicians to set up, run, and tear down each test case. At the higher levels, this information was summarized for precise management control and presentation. To measure quality, test cases were assigned pass, block, or fail status initially, with blocked tests eventually being run, passed or failed, and failed tests, when the underlying failure was addressed, becoming closed. Other information, such as the tested hardware, software, and network configuration variables, was also captured.

The final, but in many ways most important, component was the bug tracking database. This was a distributed system, with a graphical, VBA-driven, Access-based front-end available on each development team participant's desktop, and the data tables stored in a universally accessible area of the network. File read and write abilities were controlled based on the permissions of the table and the Access workgroup file, and on the user groups: developer, manager, tester, and so forth. The database was state-driven, supporting review, rejected open, assigned, test, closed, deferred, and reopened states, with appropriate ownership assigned in each state. The database also supported the generation of various reports, project metrics, and charts such as the bug summary and the opened/closed graph.

Conclusion

In this paper, we have introduced the reader to various approaches that Interactive Media Group find valuable to successfully test one of the most complex test projects the authors have worked on. From simply testing a Call Center application, to testing an integrated telephony application, to testing a CT application, to testing for data quality and reporting, to testing all these components together, we hope that our experiences are as valuable to the reader as it was to the authors.

From Unit Component Test right through to Customer Acceptance Test, the most valuable lesson learned by the authors is that inter-team communications be that by way of an issue tracking log, e-mails or telephone, is invaluable. Also, we hope that the learnings and explanations provided with respect to the Test System Fake Pipe are found as useful as we have found them to be. This is a simple approach to a very complex test project that can be used anywhere and with any tools.

---

[1] For more information on applying this technique, see *Failure Mode and Effect Analysis*, by D. H. Statamis, or, for a software-specific discussion, see *Managing the Testing Process*, by Rex Black, one of the authors of this paper.

Recommended Readings

Rex Black: *Managing the Testing Process* (1999), Microsoft Press, Seattle, WA.
Boris Beizer: *Software System Testing and Quality Assurance* (1996), International Thomson Computer Press, Boston, MA.
Kelly C. Bourne: *Testing Client/Server Systems* (1997), McGraw-Hill, New York, NY.
Roger S. Pressman: *Software Engineering A Practitioner's Approach*, McGraw-Hill, New York, NY.

Biographies

Torsten Baumann

Torsten Baumann has spent five years in the Software Testing and Quality Assurance field. He is currently the Test Team Leader at Interactive Media Group, a leading international telephony applications company, based in Toronto, Canada. Prior to this he worked as a Software Test Analyst and Team Leader of Testing/Applications Development with Speedware corporation in Montreal, Canada. His work has taken him to the U.S, Germany, Switzerland and Canada. He is currently working on a book to be titled, *Manual Testing or Test Automation: What is the correct approach?*

At Speedware corporation, Mr. Baumann was the primary responsible for the testing of an Object Class Library, a 4GL Web Development tool, as well as several Case Tools for Visual Basic and 4GL. By use of Test automation he created several test suites to cover the above-mentioned products. He recently has been working at iMG in managing and automating the testing process of a complex, multi-faceted IVR Telephony/Client Server project.

Mr. Baumann attended Concordia University's Bachelor of Commerce Program, as well as graduated from John Abbot College's Programmer/Analyst program. He is currently pursuing certification with the Quality Assurance Institute as well as his Masters of Business Administration.

Rex Black:

Rex Black has spent sixteen years in the computer industry, with twelve years in testing and quality assurance. He is the President and Principal Consultant of Rex Black Consulting Services, Inc., an international software and hardware testing and quality assurance consultancy. His clients include Dell, SunSoft, Hitachi, Motorola, Tatung, NetPliance, General Electric, Pacific Bell, IMG, Renaissance Worldwide, DataRace, Omegabyte, Strategic Forecasting, and Clarion. His work with these clients has taken him to Taiwan, Hong Kong, Japan, the U.K., Canada, Germany, Holland, France, Spain, Switzerland, and Italy, along with locations throughout the United States. He authored *Managing the Testing Process,* published by Microsoft Press in its Best Practices series.

Before becoming a consultant in 1994, Mr. Black worked as Quality Assurance Manager at Locus Computing and IQ Software, doing operating system testing for IBM and database and data warehouse query and analysis tool testing. He also spent three years as a Project Manager at XXCAL-now owned by National Technical Services-a computer testing lab. Prior to that, he worked as a programmer and system administrator.

Mr. Black holds a B.Sc. in Computer Science and Engineering from UCLA. He belongs to the Association for Computer Machinery and the American Society for Quality.

Serban Teodorescu:

Serban Teodorescu holds a Masters Degree in Electrical Engineering from the Polytechnic Institute in Bucharest, Romania from 1989 as well as a Computer Science Certificate from Herzing Institue in Montreal.

Before entering the software industry he worked in nuclear power plants automation in Romania (1989-1990) as well as Locomotive drives automation in Holland(Netherlands) (1990-1993). Here he decided to enter the computer industry as a Windows System Administrator in the Netherlands from 1991

to 1993. Mr. Teodorescu then consulted as a Database developer from 1993 to 1996 after which he became a Programming instructor in Montreal. It was at this point that he pursued his career as a Software Test Analyst with Speedware Corporation in Montreal. He is currently working as a consultant with Interactive Media Group with the Test Team.

Gordon Page:

Gordon Page has worked in the software industry for 21 years as a developer and QA/test engineer. For the past 6 years he has worked in Quality Assurance and Test for IQ Software, Intersolv PVCS, MedicaLogic, BMC Software, and as a QA consultant specializing in GUI test automation.

The authors of this paper can be reached in any of the following ways:

Torsten Baumann
Phone: +1 (905) 816 0074
Fax: +1 (416) 640 1905
E-mail: TBaumann1@compuserve.com
                Torsten_baumann@interactivemedia.com

Mail: Torsten Baumann
              905 King Street West Suite 500
              Toronto, Ontario MGK 3G9

Rex Black:
Phone: +1 (210) 696-6835
Fax: +1 (210) 696-8788
E-mail: Rex_Black@RexBlackConsulting.com
Mail: Rex Black
                      Rex Black Consulting Services
                      7310 Beartrap Lane
                      San Antonio, TX 78249

Serban Teodorescu:
Phone: +1 (416) 2636300/3403
                      +1 (416) 677-1351
Fax: +1 (416) 263-6308
E-mail: steo@netcom.ca
Mail: Serban Teodorescu
                      905 King Street West Suite 500
                      Toronto, Ontario MGK 3G9

Gordon Page:
Phone: +1 (512) 656 8069 cell
E-mail: gordonpage@earthlink.net
Mail: Gordon Page
                      10610 Morado circle #1301
                      Austin, TX 78759