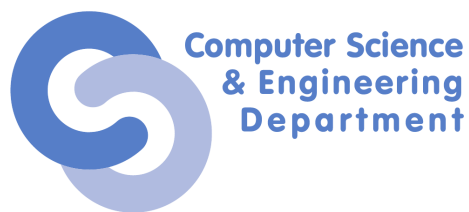


DEVELOPMENT OF A DISTRIBUTED AUTOMATED TESTING FRAMEWORK FOR EMBEDDED BASEBAND PLATFORMS

CAMELIA - ALEXANDRA GROZA



Bachelor Thesis

Computer Science and Engineering Department
Automatic Control and Computing Faculty
University POLITEHNICA of Bucharest

July 2014

Camelia - Alexandra Groza: *Development of a Distributed Automated Testing Framework for Embedded Baseband Platforms*, Bachelor Thesis, © July 2014

SUPERVISORS:

Ing. Cristian Măcăărăscu

Ș.l. dr. ing. Adrian-Răzvan Deaconescu

LOCATION:

Bucharest

TIME FRAME:

July 2014

ABSTRACT

All software users expect the applications they interact with to perform in a certain way and to complete certain tasks. In every branch of the Software Development field, extra effort is put into testing the created projects in order to assure that the users will be satisfied with the end results.

Various applications and tools come in the aid of the software developers by automating the entire testing process. Unfortunately, due to the high diversity of the field, no such tool is perfect for all software projects.

In this paper, we present a distributed automated tool aimed at helping the developers of baseband embedded projects test their products. We describe what the development process consists of, how the tool is designed and how the users interact with it.

CONTENTS

1	INTRODUCTION	1
2	TOOLS OF TRADE	3
2.1	AccuRev®	3
2.2	CodeWarrior®	4
2.2.1	Project Organization	4
2.2.2	Automation	6
2.2.3	Debugger Shell	7
2.3	Hardware Environment	8
2.3.1	The Freescale B4860 QDS®	8
2.3.2	The Networking Infrastructure	9
3	STATE OF THE ART	13
3.1	Continuous Integration	13
3.2	Development Process	13
4	ATF 1.0	17
4.1	Design	17
4.1.1	The Job Generator	17
4.1.2	The Job Worker	19
4.2	Limitations	20
5	ATF 2.0 IMPLEMENTATION	23
5.1	Design	23
5.2	The Session Configuration File	24
5.3	Workflow	27
5.3.1	The Client	27
5.3.2	The Job Generator	29
5.3.3	The Worker	30
5.3.4	The Result Aggregator	33
6	RESULTS	37
7	FUTURE WORK	39
7.1	Communicating with the Workers	39
7.2	Running a Test Session from an Archive	39
8	CONCLUSION	41
	BIBLIOGRAPHY	43

LIST OF FIGURES

Figure 1	Basic AccuRev components and commands.	4
Figure 2	RSE GUI configuration.	5
Figure 3	.cproject GUI configuration.	6
Figure 4	CodeWarrior's Debugger Shell.	7
Figure 5	B4860 Block Diagram.	9
Figure 6	The network infrastructure.	11
Figure 7	AccuRev error propagation.	13
Figure 8	B4860 Lte-L1 AccuRev structure.	14
Figure 9	ATF 1.0 design.	17
Figure 10	Development depot with Ghost Streams.	18
Figure 11	ATF 1.0 worker diagram.	20
Figure 12	ATF 2.0 design.	23
Figure 13	The Session Configuration File.	24
Figure 14	Job generator workflow.	29
Figure 15	The ATF's database.	31
Figure 16	The CodeWarrior Debugger Shell wrapper.	33
Figure 17	Worker block diagram.	35
Figure 18	Time to run a test session comparison.	38

LIST OF TABLES

Table 1	Time to run a test session comparison	38
---------	---------------------------------------	----

LISTINGS

Listing 1	Internal RSE representation	5
Listing 2	.cproject content snippet	6
Listing 3	Configuring a project with ecd.exe	7
Listing 4	Debugger Shell commands	8
Listing 5	Lars commands	10
Listing 6	Add an on demand trigger	27
Listing 7	Add a recursive trigger	28
Listing 8	List active triggers	28
Listing 9	Remove a trigger	28

ACRONYMS

SCM	Software Configuration Management
GUI	Graphical User Interface
CLI	Command Line Interface
CI	Continuous Integration
IDE	Integrated Development Environment
DSP	Digital Signal Processor
MAPLE-B3	Multi-Accelerator Platform Engine Baseband 3
RSE	Remote System Explorer
QDS	Qonverge Development System
SoC	System on a Chip
CCS	CodeWarrior Connection Server
LTE	Long Term Evolution
ATF	Automated Testing Framework
GS	Ghost Stream
SCF	Session Configuration File
UE	User Equipment

DLCCCH Downlink Control Channel

PDSCH Physical Downlink Shared Channel

PUSCH Physical Uplink Shared Channel

INTRODUCTION

In the Software Development field, the quality of a resulting product is of the essence. Through comprehensive testing, product excellency can be assured.

Software projects who follow Continuous Integration (CI) practices require the developers to perform frequent code merges between their branches. Naturally, before each such merge, validation tests need to be run to assure that functionalities weren't damaged. Thus, excessive amounts of sometimes lengthy tests are ran every day.

Automated tools are generally used to ease the testing phase. Unfortunately, no such tool is perfect for all software projects. Specifically, embedded baseband projects require a certain hardware setup that these tools aren't familiar with.

In this paper we introduce the Automated Testing Framework (ATF), one such distributed automated tool aimed at testing baseband projects.

We begin with describing environment in which this framework runs. From the platforms on which the tests run, to the Software Configuration Management (SCM) and Integrated Development Environment (IDE) used, the ATF interacts with all of them in one way or another.

We then present the framework's first version used for testing projects with precise specifications. Briefly, it polls the SCM for various events on which it triggers predefined test sessions. Each session contains multiple test that can be run by the framework's *workers* in parallel. At each session's end, a test report is sent to the developer who triggered the tests. Due to its limitations, a new version had to be implemented.

The ATF 2.0 keeps its predecessor's general design and workflow and adds some of its own components. We describe these new features starting with the Session Configuration File (SCF), the mean through which the developers are free to design their test sessions. Another important component is the *client* through which the users can interact with the framework.

By the end of this paper we list a couple of new features that can be implemented and outline the framework's performances.

TOOLS OF TRADE

Most software engineering teams require an extended list of tools to help them throughout the entire development process. From source code editing, to version control, to debugging, these applications become essential when working on comprehensive projects and, more importantly, in large teams.

Baseband engineers are by no means an exception when it comes to such instruments. In the rest of this chapter we present some of their most essential tools.

2.1 ACCUREV®

In order for a team of software engineers to be most efficient, its members need to be able to work simultaneously on the same project. As the project grows, so does the size of its development team and, most often than not, team members will be scattered geographically, thus increasing the difficulty of concurrent development.

When, inevitably, such issues arise, a [SCM](#) applications can be used. These tools, among other features, maintain a history of modifications made to shared files by each team member. Hence, the project can be easily reverted to a previous state if an adjustment needs to be undone and the persons responsible for any change can be tracked.

AccuRev® is a [SCM](#) application used by baseband engineers to control the concurrent development process. It allows developers to work privately, share code, develop code serially, lock files if necessary, and protect a code base. Its basic components are: ¹

- *Workspace* - the private local development area of each user.
- *Stream* - shared configurations of related elements which change over time; code can be promoted into or inherited from streams.
- *Depot* - the main repository on a server for all related source code.
- *Snapshot* - static (protected) stream that cannot be moved, renamed, or altered.

[Figure 1](#) portrays the fundamental AccuRev components and commands. Streams and workspaces are organized in a tree-like manner with the former as leaves. Workspaces stand behind streams with which they synchronize their contents. A classic scenario is one in

¹ AccuRev® Quick Reference

The root stream usually contains the latest functional code and is called Integration stream.

which multiple developers each have a workspace behind the same stream, stream used for collaboratively developing the same feature.

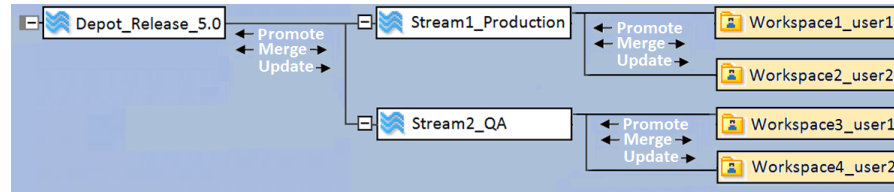


Figure 1: Basic AccuRev components and commands.²

When changes from a local workspace need to be integrated with others' work, they are *promoted* to the workspace's backing stream. At this point they belong to the stream's *default group*³ from where they can be promoted again to the stream's parent and so on until the root stream is reached. If two streams want to promote versions of the same file to a common backing stream, a *merge* is required between the two file versions. If a user wants to retrieve in his own workspace the changes made by one of his colleagues, he has to *update* his workspace, thus fetching all changes from the backing stream. Changes propagate downward automatically through streams. This means that, opposed to workspaces, all child streams will immediately receive the changes once a promotion to a parent stream occurs.

Beside a Graphical User Interface (GUI) used by most developers, AccuRev also has an equivalent and elaborate Command Line Interface (CLI) useful in automated contexts.

2.2 CODEWARRIOR®

The CodeWarrior® IDE provides an efficient and flexible software-development tool suite. It is used for the creation of projects that run on a number of embedded systems.⁴

CodeWarrior is based on the Open Source Eclipse IDE.

All CodeWarrior versions used by baseband engineers contain, among other features, a wide array of tools developed especially for the *Star-Core Digital Signal Processor (DSP)* and the *Multi-Accelerator Platform Engine Baseband 3 (MAPLE-B3) Hardware Accelerator*.

2.2.1 Project Organization

Following the Eclipse's approach to organizing resources, each CodeWarrior project contains, aside from its source code, a set of configuration files through which it passes certain settings to the IDE. We will describe two of these file types:

² AccuRev Quick Reference

³ files that differ from their version in the backing stream

⁴ CodeWarrior Common Features Guide

1. A set of XML files used for configuring the Remote System Explorer (RSE) - the connection to the simulator / hardware device on which the project will run. In Figure 2 we show how these options can be set through the IDE's GUI.

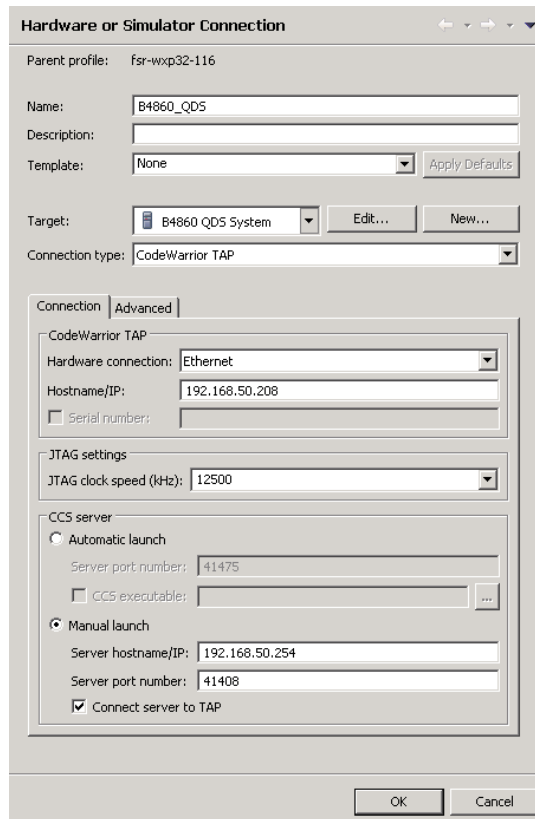


Figure 2: RSE GUI configuration.

In Listing 1 we see how some of these settings are represented in the XML RSE configuration files.

Listing 1: Internal RSE representation

```
...
<property key="propertySet.[cw.dbg.ct.cwtap.jtag].
  debugConnection" value="Ethernet"/>
<property key="propertySet.[cw.dbg.ct.cwtap.jtag].hostname"
  value="192.168.50.208"/>
<property key="propertySet.[cw.dbg.ct.tap.jtag].
  chainSpeedTCK" value="12500"/>
<property key="propertySet.[cw.dbg.ct.ccs].CCSIPAddress"
  value="192.168.50.254"/>
<property key="propertySet.[cw.dbg.ct.ccs].
  remoteServerPortNumber" value="41408"/>
...
```

2. *.cproject*, the main XML project configuration file used for storing information regarding each available build target. Some of the informations stored in such files are various build flags, lists of defined and undefined preprocessor macros and names of resulting binaries. Similar to the [RSE](#) settings, these configurations can be set through the [IDE's GUI](#), as we can seen in [Figure 3](#).

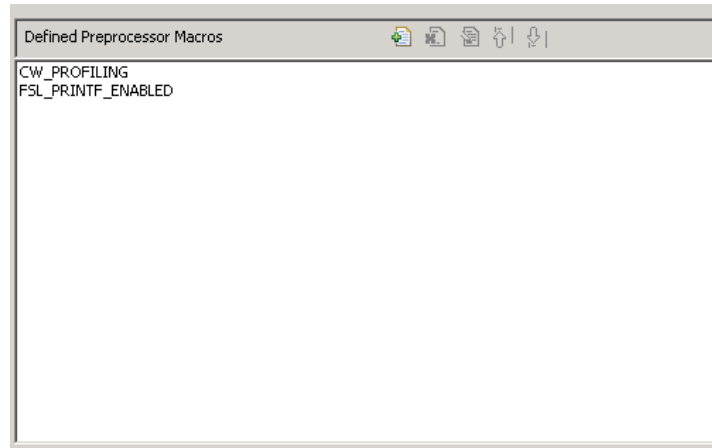


Figure 3: *.cproject* GUI configuration.

Again, in [Listing 2](#) we can see how some these settings are portrayed in the *.cproject* file.

Listing 2: *.cproject* content snippet

```
...
<option id="..." name="Defined Preprocessor Macros"
  superClass="..." valueType="...">
  <listOptionValue ... value="CW_PROFILING"/>
  <listOptionValue ... value="FSL_PRINTF_ENABLED"/>
</option>
...
```

Furthermore, when opening and working with a project, the CodeWarrior [IDE](#) needs a *workspace* in which to store information regarding the current session. The workspace contains a *.metadata* directory in which [IDE](#) settings are kept along with configurations of the currently opened projects. Rebuilding the project before launching it, displaying certain pop-up messages and using cached [RSE](#) settings are just part of the metadata kept in the [IDE's](#) workspace.

2.2.2 Automation

Aside from the [GUI](#), the CodeWarrior [IDE](#) also provides a command-line tool, *ecd.exe*, which can be used to configure and build projects.

For building a project and loading it into a workspace, a command as the following can be used.

```
> ecd.exe -build -data workspace_path -project project_path -
    config target
```

As for managing the [RSE](#) system settings or the launch configurations, the `-setOptions` and `-getOptions` flags can be used. The `-getOptions` flag retrieves the project's current settings in a key-value format. In a similar manner, the `-setOptions` flag can be used to update the value of a provided option key. In [Listing 3](#) we exemplify how we can obtain the list of defined preprocessor macros for the B4860_QDS_rev2 configuration of a given project and how the `-Wall` compile flag can be activated for the same project.

Listing 3: Configuring a project with ecd.exe

```
> ecd.exe -getOptions -project project_path -config B4860_QDS_rev
    2 -option scc.preprocessor.definedMacros
configuration(B4860_QDS_rev2):
    scc.preprocessor.definedMacros = FSL_PRINTF_ENABLED
    memcpy=memcpy_SBL1_opt
!Command.Success!

> ecd.exe -setOptions -project project_path -config B4860_QDS_rev
    2 -set scc.compiler.reportAllWarnings scc.compiler.
    reportAllWarnings
!Command.Success!
```

2.2.3 Debugger Shell

The debugger is another one of CodeWarrior's features that supports a command line interface. Various debug commands can be executed through a TCL debugger shell embedded in the [IDE](#), as seen in [Figure 4](#).

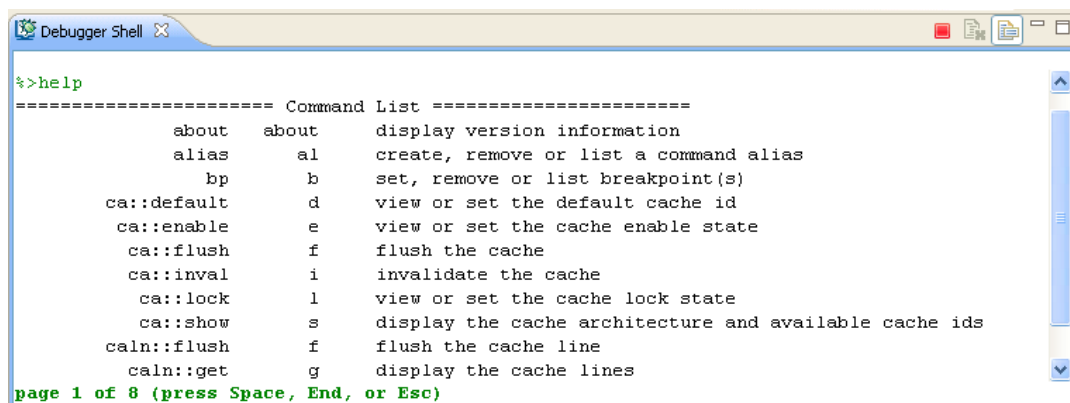


Figure 4: CodeWarrior's Debugger Shell.

In Listing 4 are some of the basic Debugger Shell's commands used by baseband engineers that are unique to CodeWarrior.

Listing 4: Debugger Shell commands

log	log Commands and/or Session to file
debug	launch a debug session
bp	set, remove or list breakpoint(s)
go	start a thread
kill	close the specified debug session(s)

The entire debug process can be easily automated by passing a TCL script to the IDE when calling it. After executing the following command, a new instance of CodeWarrior will open and start running debug_script.tcl in its Debugger Shell⁵.

```
> cwide.exe -clean -vmargsplus -Dcw.script=debug_script.tcl
      -data workspace_path
```

2.3 HARDWARE ENVIRONMENT

The most important tools used by the baseband engineers are the hardware boards on which they run their code. These boards are designed for multi-standard wireless base stations and are used by different clients from the wireless telecommunications field.

In the following subsections we will present the architecture of such a board (the Freescale B4860 QDS[®]), the interface between the CodeWarrior IDE and the hardware equipment, the networking infrastructure and the necessity for a board management tool.

2.3.1 The Freescale B4860 QDS[®]

Freescale develops a pool of QorIQ Qonverge platforms which combine Power Architecture cores with StarCore DSPs for packet and baseband processing, security and more.

The B4860 Qonverge Development System (QDS) is one such flexible platform that supports the B4860 baseband System on a Chip (SoC) device, providing a complete application development environment. The board is intended for macrocell base stations in wireless infrastructures as well as aerospace and defense applications.

The B4860 SoC, described in Figure 5, contains ⁶:

- Four dual-threaded e6500 cores built on Power Architecture technology up to 1.8 GHz with AltiVec 128-bit SIMD engine

⁵ The project that will be debugged needs to be loaded in the used workspace before calling the IDE. This is easily achieved by the ecd.exe tool when building the project beforehand (see Section 2.2.2).

⁶ http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=B4860

- Six SC3900FP Fixed/Floating-point **DSP** cores built on StarCore technology up to 1.2 GHz
- **MAPLE-B3** baseband acceleration processing engines for LTE, LTE-Advanced and WCDMA (HSPA/HSPA+)

QorIQ Qonverge B4860 Block Diagram

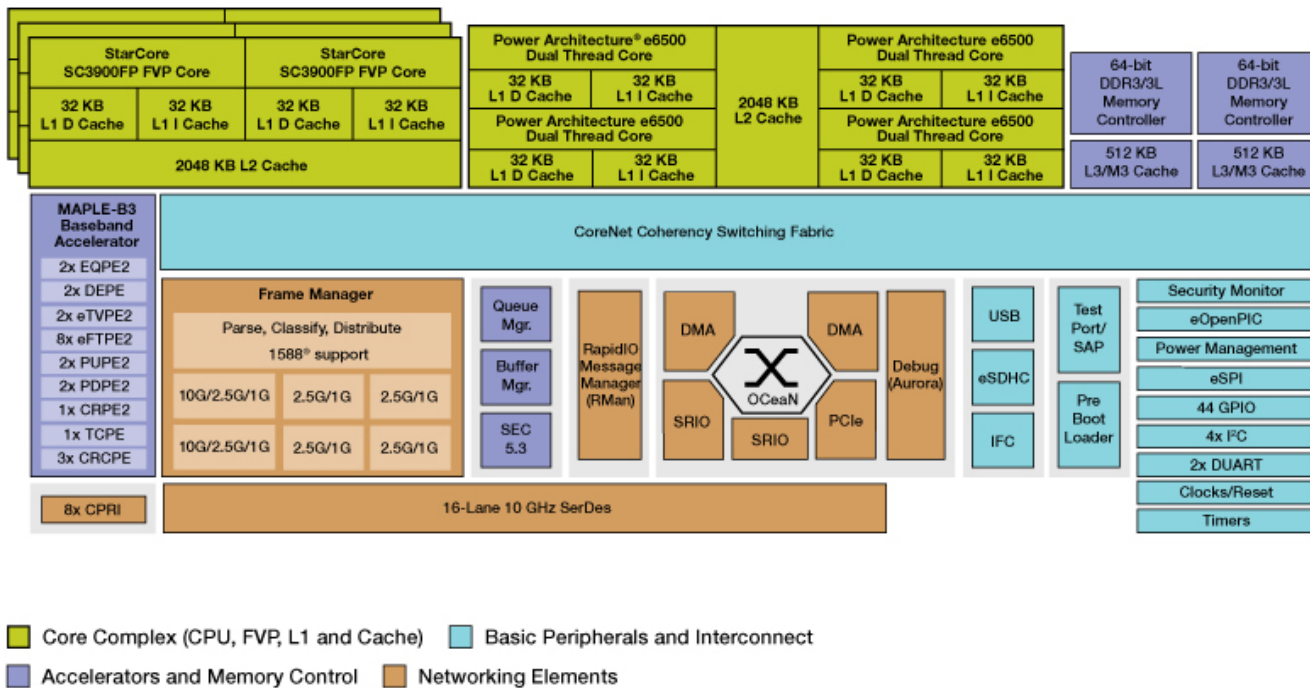


Figure 5: B4860 Block Diagram.⁷

System debugging is enabled on the B4860 **QDS** platforms by the CodeWarrior TAP which connects the board's standard debug port (JTAG) to a developer's workstation via Ethernet. The CodeWarrior tools communicate with the TAP run controllers through a CodeWarrior Connection Server (**CCS**) software module (**CCS** processes run on the boards awaiting connections from the **IDE**).

2.3.2 The Networking Infrastructure

Due to their complex architecture, boards such as the B4860 **QDS** are fairly expensive. Thus, in teams of 10 - 15 engineers, the budget does not permit for each developer to have his/her own platform, on his/her desk, to work on. In order for everyone to be able to debug their projects on such platforms, all boards have been integrated in a private network to which all engineers have access. Hence, the

⁷ http://cache.freescale.com/files/graphic/block_diagram/B4860_BD_IMG.jpg

developers can use any one of the boards from the board pool. The network infrastructure is portrayed in [Figure 6](#).

Developers outside the private network do not have access to the board pool. For this reason, an additional machine is placed with two interfaces: one for the private network and one for the outside. This machine is responsible for mediating the [CCS](#) connections between the developers and the boards. It runs a set of [CCS](#) processes, one for each board, to which the engineers connect through the CodeWarrior [IDE](#). In turn, these processes connect to the [CCS](#) process running on the appropriate board and forward the packages between the two.

When multiple developers share the same resources, synchronization issues are bound to occur. In order to manage the access to the board pool, a tool named Lars was introduced. This command line application runs on a separate Linux virtual machine from inside the private network and it allows users to *reserve* and *release* boards before and after using them. Thus, a scenario in which two developers debug on the same board and overlap is less likely to happen. The main commands are described in [Listing 5](#). Lars is written in Python and uses a PostgreSQL database to keep all board related information.

Listing 5: Lars commands

<code>list boards</code>	list all the boards in the system along with their reserved status and a summary of their configurations
<code>info <board></code>	list more detailed board configurations such as board revision, processor revision, the IP and the port of the board's CCS process running on the mediator machine, the board's IP, etc
<code>reserve <board></code>	reserve the board
<code>release <board></code>	release the board
<code>power <board></code>	power the board
<code>reset <board></code>	reset the board
<code>restart <board></code>	software reset the board

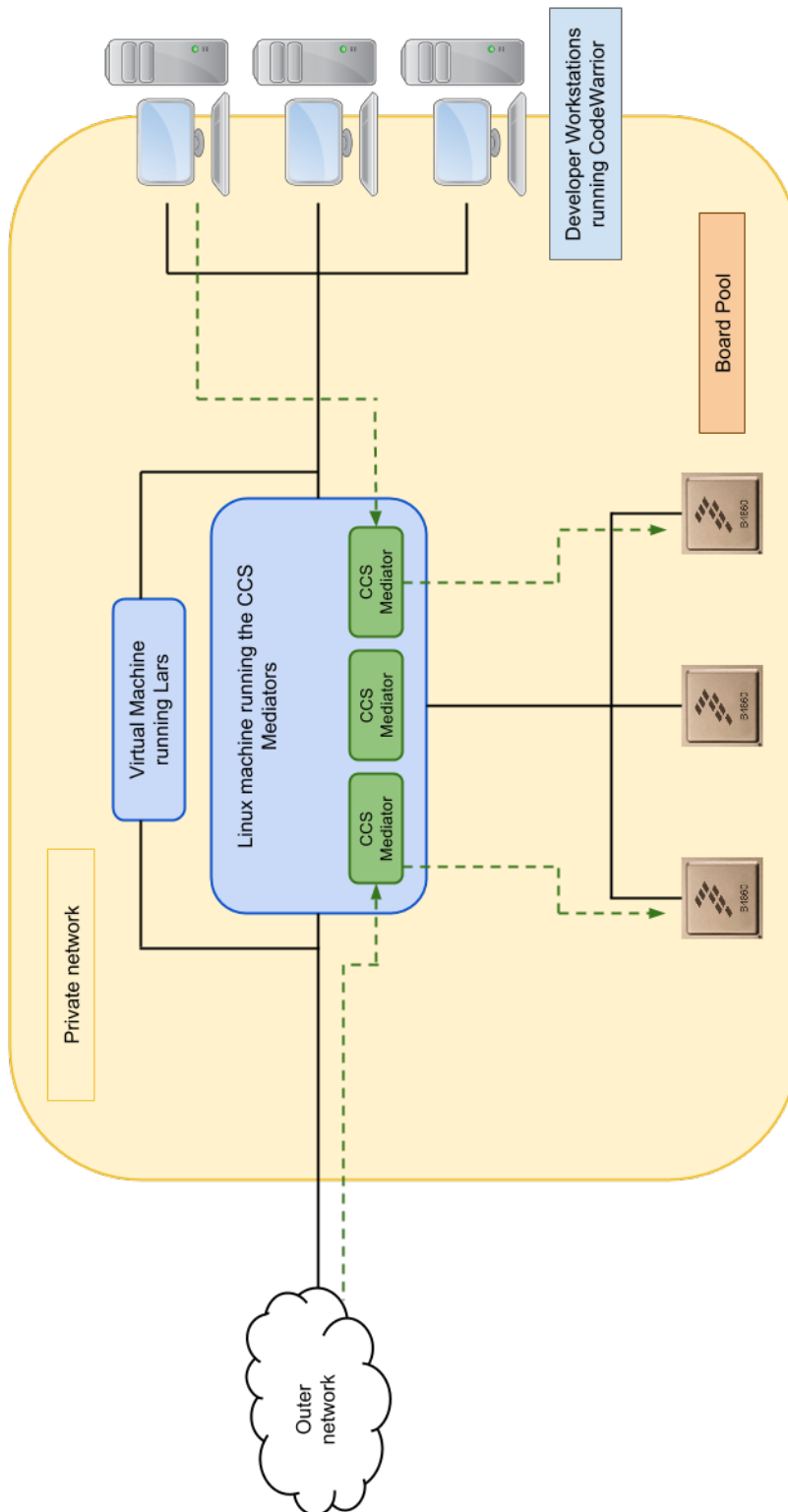


Figure 6: The network infrastructure.

STATE OF THE ART

Project management is a requirement in large teams of software developers. In this chapter we will describe the development process in a team of baseband engineers and justify the need for a testing framework.

3.1 CONTINUOUS INTEGRATION

CI is the software development practice of merging a team's work frequently, usually on a daily basis [1]. Throughout this process, all project components are integrated from the start, instead of extra effort being put into the merging process before a milestone/release.

One of the main issues that occur when Continuous Integration practices are being used is that, due to the high number of promotions a day, error prone code may be easily integrated in the project. Furthermore, in AccuRev (Section 2.1), once a change is promoted to the Integration stream, it will automatically propagate downward into all the other streams. Thus, once an error finds its way to the root of the depot it will affect the entire project. Figure 7 portrays this scenario. In order to avoid this issue, it is a good practice to run unit tests in the local workspace before any promotions to the Integration stream.

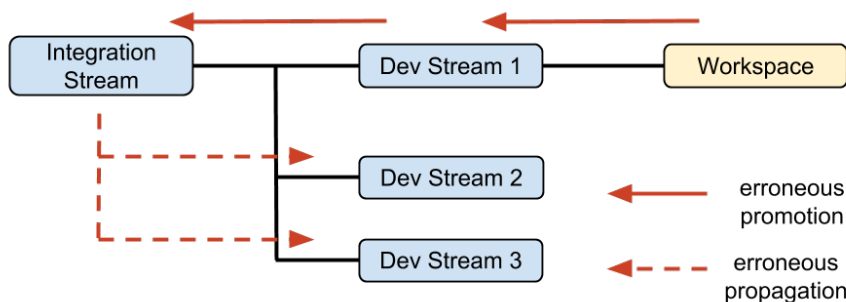


Figure 7: AccuRev error propagation.

3.2 DEVELOPMENT PROCESS

In this section we will describe the development process for a team of baseband engineers by taking a look at the B4860 Long Term Evolution (LTE) L1 project.

The team's goal is to implement the [LTE](#) standard on top of the B4860 [QDS](#) platform. The project is split up into multiple libraries that one or more developers work on. For each such component, a separate AccuRev stream is created. Clients require separate components instead of the entire project, so each stream acts as an Integration stream on its own. For each of the component's features a new stream can be created with multiple developer workspaces behind it, as show in [Figure 8](#).

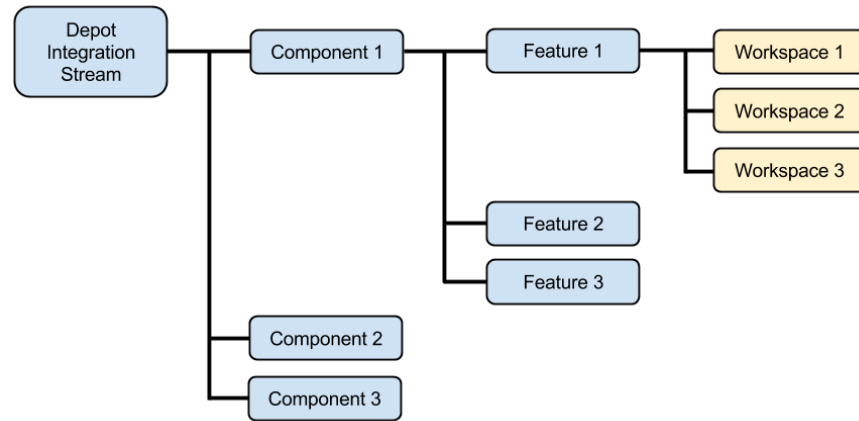


Figure 8: B4860 Lte-L1 AccuRev structure.

In order to assure the quality and correctness of their code, the developers maintain sets of *unit tests* and *functional tests*. Unit testing refers to testing program units in isolation. However, there is no consensus on the definition of a unit. Some examples of commonly understood units are functions, procedures, or methods. In functional testing, a program P is viewed as a function that transforms the input vector X into an output vector Y such that $Y = P(X)$ [3].

Following the [CI](#) practices, the developers run a set of small preliminary tests before promoting features from their workspaces. These batches are called *smoke tests*. In addition, when major patches or configurations are introduced, a wider set of tests, called *regression tests*, is executed.

Test sessions such as these can take up to 8 hours to execute. Furthermore, these tests require excessive processing power, thus the developer being forced to wait for the sessions to finish until his/her workstation will be available again. Due to this limitations, the engineers are required to leave the test session to run over night or over the weekends.

Unfortunately, this solution causes other issues to arise. Due to the fact that the developers aren't at their computers when the tests run, errors caused by the networking environment or the tools' limitations can not be avoided or resolved on the spot and the entire test session goes to waist.

Such errors occur fairly often and may be caused by one or more of the following:

- One of the CCS processes running on the mediator machine may be in an unreliable state. This issue will cause the CodeWarrior's Debugger Shell to throw certain error when trying to connect to a board and can be fixed by restarting the respective CCS process.
- The board may be in an unstable state. Again, this will cause the CodeWarrior's Debugger Shell to throw errors when attempting to debug the project. This issue can be fixed by restarting the board (either through hardware or software mechanisms).

In order to overcome the issues described in [Section 3.2](#) and to ease the overall testing process, an ATF was implemented. In the following sections we will discuss its design and its limitations, that in turn led to the development of its successor, the ATF 2.0.

4.1 DESIGN

At its beginning, the ATF was needed for testing only one baseband project with precise specification. Its goal was to assure that all the component streams and the Integration stream contained only valid functional code by automatically launching predefined test sessions on AccuRev promotions. Furthermore, through interacting with AccuRev, the developers were able to trigger other test sessions. The ATF would execute these session on multiple machines and would send e-mail test reports at the end.

The framework's main components ([Figure 9](#)), discussed in the following sections, were:

- the job generator, responsible for triggering test sessions on AccuRev promotions or user interactions by adding jobs to the job queue.
- the job workers who executed the actual tests by extracting jobs from the job queue.

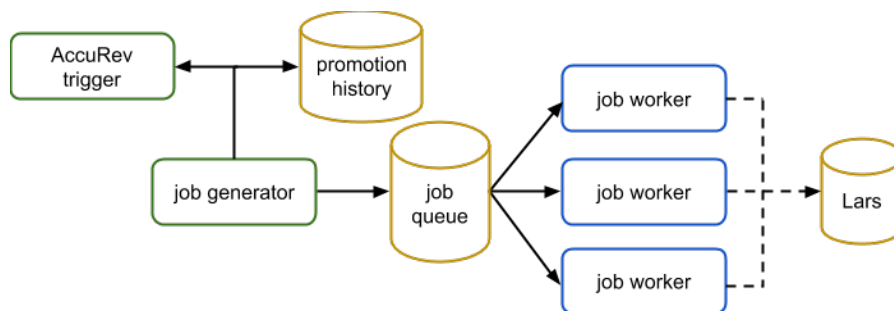


Figure 9: ATF 1.0 design.

4.1.1 The Job Generator

The ATF comes in the aid of the engineers who follow the CI practices. Since the developers may not have enough time or resources to run

tests before integrating their changes with their colleagues', they rely on the ATF to keep all essential streams *clean* from any bugs. This is accomplished by creating a special stream, named Ghost Stream (GS), for each stream of interest, as pictured in Figure 10.

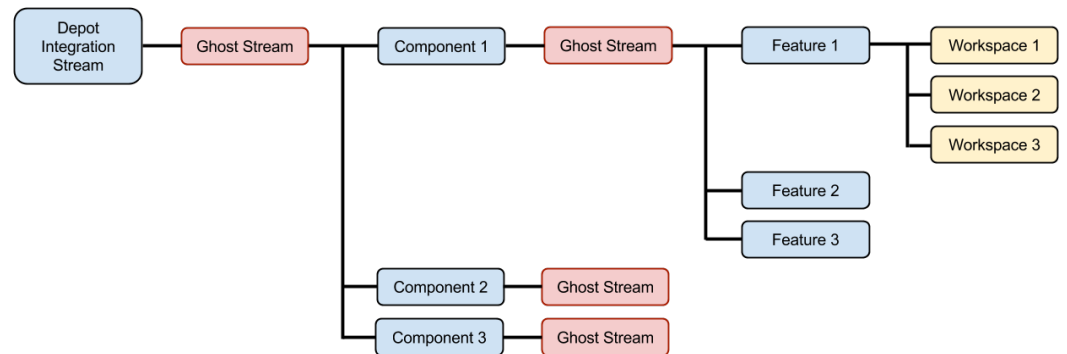


Figure 10: Development depot with Ghost Streams.

A GS is responsible for guarding the stream it precedes from promotions containing erroneous code. The ATF is aware of all the GSs in the depot. For each GS, it knows which tests it needs to run when changes appear on it. If the tests pass, the framework will automatically promote the changes to the parent stream. If not, the developer who promoted the changes in the first place will be notified that his/her code contains bugs and the framework will wait for the next promotions.

In order to detect the changes that occur on the GSs, the ATF polls the project's AccuRev depot every minute: it uses an AccuRev command to see the latest promotions to each GS and it launches the corresponding test sessions. To differentiate between the promotions that were already taken into account and the ones that weren't, the framework maintains a history of these events in a PostgreSQL table.

The ATF knows which tests to run for each GS depending on the stream's name. Thus, all streams need to respect a pattern such as the following:

```
<depot name>_<component name>_GS_<anything>
```

In addition, in order to trigger a test session without promoting to a GS, the developers can create temporary streams (snapshots) with the following name pattern for launching a Test On Demand (TOD). The snapshot would be deactivated automatically by the framework.

```
<depot name>_<component name>_GS_TOD_<anything>
```

Based on the *component name* specified in the stream's name, the ATF adds the tests that need to run to a test job queue, in the form of a PostgreSQL table, by specifying:

- the stream on which the test will run

- the job's status (new / running / finished)
- the platform on which the test will be executed (predefined value for each component)
- the ID of the developer who launched the test session
- the test's resolution
- the machine on which the test runs

4.1.2 *The Job Worker*

Each machine that is part of the ATF's environment polls the job queue looking for new jobs to run. For each job it finds, it obtains certain predefined files from the job's stream and searches the directory structure for a specific path that points to the CodeWarrior project configuration files. An error is reported if the file structure does not follow the expected directory structure.

Once the CodeWarrior project configuration files are found, the build process begins. The ATF uses the automated tools described in [Section 2.2.2](#) to build the project with a predefined build target.

If the build succeeds, the framework will build in a similar manner a predefined test project written by the developers¹. For the test's actual execution, a platform on which to run is necessary.

To use a board from the board pool, the ATF needs to reserve one in the name of a generic user by directly accessing the PostgreSQL database used by Lars in order to void race conditions.

After reserving a board, the framework extracts the board's configurations from Lars' database and modifies the project's CodeWarrior configuration file (*.cproject*) so that the IDE will use the reserved board's RSE settings instead of the default ones when debugging (see [Section 2.2.1](#)).

The next step in running the test is preparing the required test vectors. These vectors are the test's input. The values returned by the component after processing these vectors are compared to a set of reference values. This is how the test's resolution is set. The preparation of the vectors is again the responsibility of the ATF.

The last step in the test's execution is the IDE's actual launch. As mentioned in [Section 2.2.3](#), the debug process can be controlled through the CodeWarrior's Debugger Shell. This shell can be manipulated through a TCL script passed as parameter when starting the IDE. In its first version, the ATF was responsible for managing this script. At the test's end, the debugger script returns a pass/fail status which the ATF interprets. Furthermore, if unexpected errors arise during the

¹ When the ATF 1.0 was used, the components were in fact libraries. Thus, they required separate test projects to link and test them.

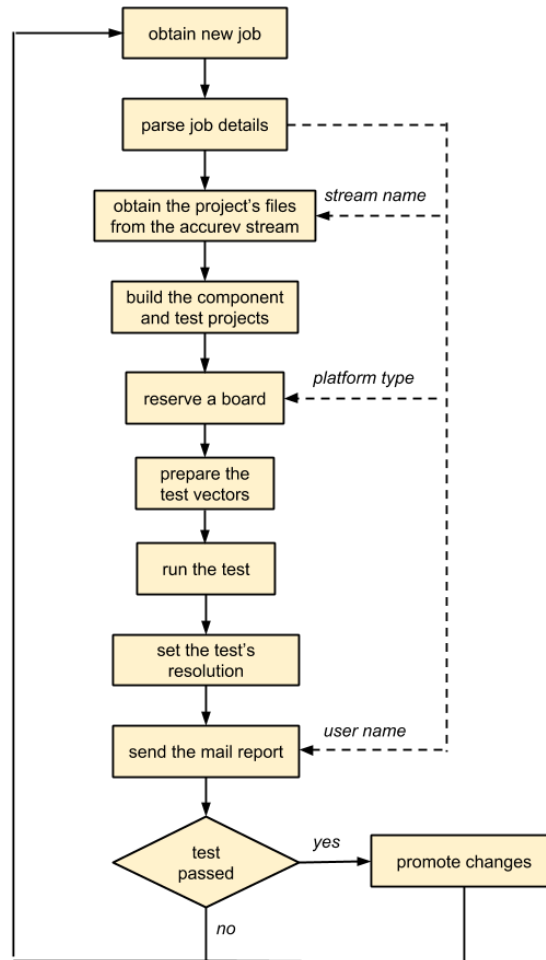


Figure 11: ATF 1.0 worker diagram.

test's execution caused by the involved tools' limitations, the job is re-launched.

This entire test execution process is portrayed in [Figure 11](#).

4.2 LIMITATIONS

As described in the previous section, the first ATF was working with an excessive set of predefined configurations. For many of these, if the engineers needed a different structure, the framework's administrator had to be contacted in order to patch the exceptional cases. Below are listed some of the major limitations encountered:

- The framework was written in TCL, a scripting language with a long history but with no built-in support for Object Oriented syntax, without data structures, and with a relatively small active community.

- Only one project depot was supported. This wasn't relevant in the beginning when only one project needed the ATF but when more projects appeared, they were difficult to integrate in the framework.
- The GS names had to indicate the component that needed testing.
- AccuRev uniquely-named snapshots were created in order to launch test sessions on demand but once the test ran, the snapshot would remain in the depot. Once the snapshot was deactivated by the ATF, its name still couldn't be used again by other developers. Thus, the depot would in time be flooded with deactivated snapshots.
- For each component, a predefined set of source files would be extracted from AccuRev when running a test. If additional source files were introduced, the ATF's administrator had to be contacted in order to update the component's required files list.
- The project had to follow a specific directory structure when it came to organizing its source files and configuration files.
- Each test assumed that a library had to be built and a test had to be run. If a project required only library building, the ATF's administrator had to intervene and add the project to a special build-only list.
- The tests always ran on the same platform types. At the time, those were the only options but as more projects appeared, so did more platforms.
- The build target was presumed to be first one specified in the project's CodeWarrior configuration file for both the libraries and the tests.
- The ATF's administrator was in charge of managing the TCL Debugger Shell script.

ATF 2.0 IMPLEMENTATION

The first ATF was a simple solution for a simple problem but as multiple projects appeared with different configurations and requirements, patches and exceptions had to be introduced in order to solve the new, more complex, issues. Overtime, maintaining the project became too big an effort due to the high number of irregularities. The chosen solution was to implement a new, more robust framework that supported easily configurable projects.

5.1 DESIGN

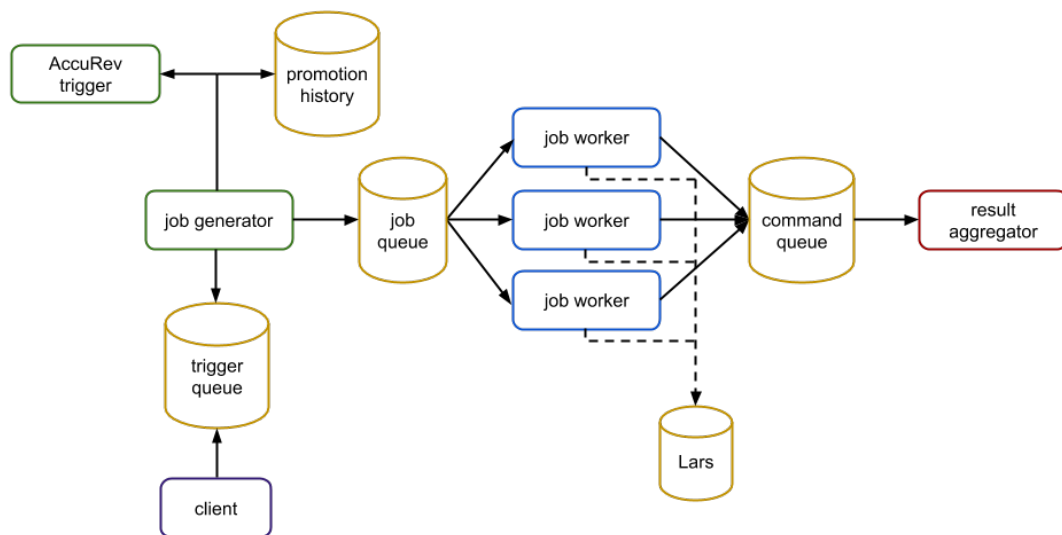


Figure 12: ATF 2.0 design.

The ATF 2.0 introduces several changes and new features from its predecessor:

- It is written in Python, a scripting language with a comprehensive standard library, Object Oriented syntax support and a large active community of both users and developers.
- A client through which users can trigger test sessions instead of using AccuRev snapshots. The sessions can be launched either at the moment or multiple times with a recurrence factor specified through a cron syntax.
- A SCF maintained by the user through which they are free to define and customize their test sessions without the need of contacting the framework's administrator (see Section 5.2).

- A result aggregator in charge of all the actions that need to be performed at the end of a test session (obtaining all the results from the workers, composing the final mail report and promoting the changes to the parent stream in the case of a [GS](#)).
- A command queue through which various instructions can be passed on to all the main components.

In [Figure 12](#) we present the framework's new design. We will discuss its updated workflow in the following sections.

5.2 THE SESSION CONFIGURATION FILE

The [SCF](#) is a XML document maintained by the [ATF](#)'s users in their projects' depots. Each [SCF](#) describes the test sessions that can be ran for the provided project.

With the help of this file, the developers can split a single large test into multiple shorter ones that can be ran in parallel by the [ATF](#), thus shortening the test's duration considerably.

These files contain information regarding the platforms on which the tests will run, the CodeWarrior versions that will be used, the paths to the tested projects, their build targets, macro definitions and dependencies, plus a set of pre-processing and post-processing scripts. Furthermore, the TCL debugger script is maintained by the developers and the [ATF](#)'s only responsibility is calling it.

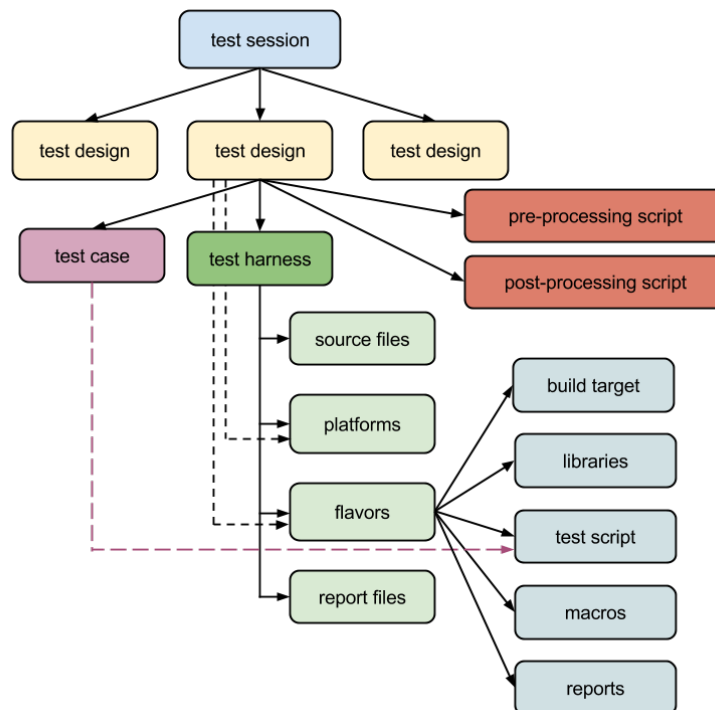


Figure 13: The Session Configuration File.

The SCF's structure is pictured in Figure 13. It was designed to follow the "BS7925-2 Standard for Software Component Testing" [2] and its main components are:

- *The test session*, the main entity used to test any number of project with different configurations. It is launched by the users by interacting with either AccuRev's GSs or with the ATF's client and it contains multiple test designs.

```
<test_session name="DLCCH_PROC_session">
  <test_design name="DLCCH_PROC_Set1"/>
  <test_design name="DLCCH_PROC_Set2"/>
</test_session>
```

- *The test design* selects the actual test environment that will run (the test harness), picks its configuration (its flavor and platform) and specifies its input (the test case). Furthermore, the users can add a set of pre-processing and post-processing scripts.

```
<test_design name="PDSCH_AL_List5_Sync">
  <test_harness name="module_name"
    flavor="rev2_sync_build"
    platform="B4860"/>

  <test_case>ATF\TCnameList5</test_case>

  <pre_processing what_to_call="ATF\pre.py"
    from_where_to_call_it="L1\SP\"/>

  <post_processing what_to_call="ATF\post.tcl"
    from_where_to_call_it="L1\SP\"/>
</test_design>
```

- *The test case*, the input data used by the running test. By splitting the initial input vectors into multiple sets, the tests take shorter time to end when an in parallel.
- *The pre/post-processing scripts* will be run before/after the test's execution from relative paths specified by the user.
- *The test harness* contains the descriptions of the test's environments. It contains the source files needed to run the test, a series of flavors describing different configurations and a series of platforms on which it can run.

```
<test_harness name="MatrixIC_test">
  <source_files>ATF/MIC</source_files>

  <flavors>
```

```

    <flavor name="rev2_build">
      <build_target>B4860_QDS_rev2</build_target>

      <test_script what_to_call="ATF\MIC\run.tcl"
        from_where_to_call_it="L1\SP"/>

      <reports>L1\doc\TestReport</reports>

      <dependencies>
        <component name="lib" flavor="rev2"/>
      </dependencies>

      <macros>
        <macro name="M_EDF" defined="true"/>
      </macros>
    </flavor>
  </flavors>

  <platforms>
    <platform type="B4860">
      <project_path>L1\SP\project\CW</project_path>
    </platform>
  </platforms>
</test_harness>

```

- *Harness platforms* that specify the path to the test's CodeWarrior project.
- *Harness flavors* that describe the test's possible configurations. They specify the test's build target, the preprocessing macros that need to be activated and/or deactivated, the components (libraries) that it depends on, the TCL test script and the report files that will be sent back to the user at the test's end.
- *The components* are projects that need to be built before the test. They have a similar structure to that of the test harness (they contain an enumeration of flavors and platforms) but they lack the source files, the test script and the dependencies.

```

<component name="module_lib">
  <flavors>
    <flavor name="rev2lib">
      <build_target>lib</build_target>
    </flavor>
  </flavors>

  <platforms>

```

```

    <platform type="B4860">
      <project_path>L1\SP\CW</project_path>
    </platform>
  </platforms>
</component>

```

- *The test script* is the TCL CodeWarrior Debugger Shell script that is responsible for running the test. The ATF calls it from a user-specified path and passes it the *test case* mentioned in the *test design*.
- *Hardware descriptors* are a separate section of the SCF and they specify the CodeWarrior versions that need to be used by all the platform defined in the test harnesses.

```

<hardware>
  <platform name="B4860next"
    type="B4860"
    cw="10.8.0"/>
</hardware>

```

- *Platform tags* can be added to the test session or in the flavors of harnesses and components to limit the range of boards that can be used to run a test. For example, such tags may contain board revision numbers or processor revision numbers. These tags are searched for by the ATF in the output of the *lars list boards* and *lars info* commands.

5.3 WORKFLOW

5.3.1 The Client

The client is the framework's component with which the users interact most often. It is a script placed on the Virtual Machine running Lars (see Figure 6) so that all the developers can access it. For each project supported by the ATF there is one client instance. Through it they can trigger test session on demand or recurrent sessions.

The developers pass to the client script the name of the test session they want to run, the name of the stream on which the session will be executed, and the recurrence factor (either *-now* for on demand test or *-cron <cron>* for recurrent tests). They can also specify the users who should receive the test reports at the end of the session. If no users are specified, the user who ran the command is registered. In Listing 6 and Listing 7 we can see how the users interact with the script in order to trigger test sessions.

Listing 6: Add an on demand trigger

```
$ ./atf_client.py add -stream Feature -session TestSession -now -coreid user1
```

Listing 7: Add a recursive trigger

```
$ ./atf_client.py add -stream Integration -session IntSession -cron 0 0 * * *
The next two iterations will be at (UTC +2):
2014/06/27 00:00:00
2014/06/28 00:00:00

Continue?
[y/n]:
```

The provided values are introduced to the Trigger Queue, a PostgreSQL table with the fields described in [Figure 15a](#). The *Depot* field is set by the script itself from its configurations. The *Cron* field is set to either *cron* or *now*, depending on the trigger's type. Similar, the *NextCron* field is set to either *now* or the timestamp of the trigger's next iteration. This timestamp is calculated using the *croniter* Python module ¹.

The users can also list ([Listing 8](#)) and remove triggers. By removing a trigger, its *NextCron* field is set to *done*.

Listing 8: List active triggers

```
$ ./atf_client.py list -triggers
```

Users	Stream	Session	Cron	Next iteration
user_1	Integration	ModuleSession_next	0 21 * * 6	2014/06/21 21:00:00
user_2	Integration	LTE_regression_next	30 1 * * 1	2014/06/19 01:30:00

Listing 9: Remove a trigger

```
$ ./atf_client.py remove -trigger 377
The following trigger will be deactivated:

ID:      377
Users:   user_1
Stream   Integration
Session  LTE_regression_session_next
Cron     30 1 * * 1
Next iteration (UTC +2):      2014/06/27 01:30:00

Continue?
[y/n]:
```

¹ <https://pypi.python.org/pypi/croniter/0.3.4>

5.3.2 The Job Generator

The job generator has a somewhat similar workflow compared to its predecessor. As shown in [Figure 14](#), multiple processes run in parallel waiting for different types of events. A main process spawns a trigger polling process and a GS polling process for each project which require such streams.

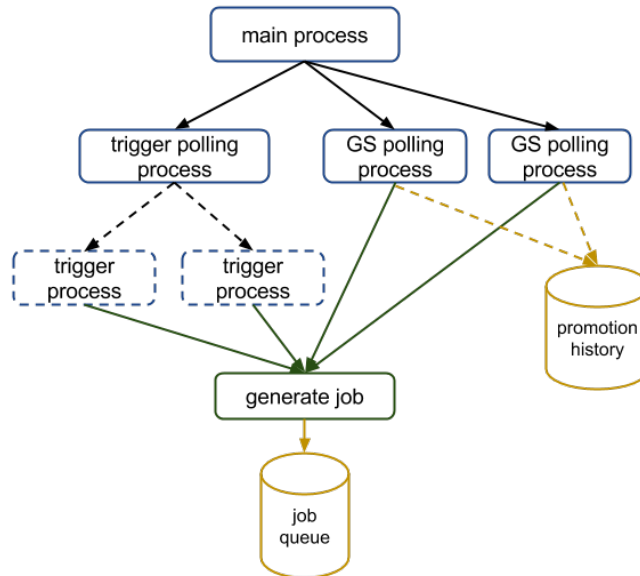


Figure 14: Job generator workflow.

The trigger polling process is running in an infinite loop and checks the Trigger Queue for new events. It first obtains all the triggers that aren't *done*. For each one of them, if they are on demand triggers (their *cron* field is set to *now*) it sets them as *done* and spawns a new process that will be responsible for the actual job generation. If the triggers are recursive ones, it updates their *NextCron* field and again spawns a new process for the job generation.

The GS polling processes also run in a loop and search for the latest promotion on all the streams that contain a certain predefined pattern and don't have an empty *default group* (see [Section 2.1](#)). This pattern is set when a new project is added to the ATF by the project's maintainer. For each such stream, the latest promotion's transaction ID is verified and searched for in the Promotion History table. If it already exists, the transaction was already taken into account. If not, the stream's status is verified. If it contains conflicts it is abandoned. Otherwise, the job generation routine is called. The Promotion History table's format is described in [Figure 15b](#).

The actual job generation obtains the SCF from the stream (provided by the user through a trigger or a GS) and parses it in order to find the description of the test session it needs to launch. As far as GSs are

concerned, the ATF searches for a session that has the same name as the stream. For triggers, the users provides the name of the session.

The ATF does some lexical and semantic checks in order to verify the SCF's structure. If, for example, two sessions have the same name, or a test design doesn't exist, or a platform uses a CodeWarrior version that the framework doesn't support, the job generation is abandoned and the user is notified.

If the SCF passes all these checks, the test session's test designs are considered separate jobs and inserted in the Job Queue for the workers to run them. The queue's structure is pictured in Figure 15c.

In order to avoid other developers from promoting to the testes streams and contaminating the used code between two jobs of the same session, the framework saves the AccuRev transaction on which to run the test. Thus, when obtaining the source files from the stream, all workers will use the same transaction. For GSSs, the transaction ID is the ID of the promotion which triggered the session. For user triggers, the current timestamp is used instead of an ID.

Furthermore, for GSSs, if the test session passes, the modified files from the stream need to be promoted to the parent stream. Thus, a promotion flag is added to the Job Queue which indicates if promotion is requested at the end of the session. Naturally, this flag is set to *False* for user triggers and to *True* for GSSs.

The job's *status* field, initially set to *new*, is updated by the workers along with the *resolution* and *machine* fields.

After adding the new jobs to the Job Queue, the framework sends an e-mail to the users who launched the session to let them know the details of the jobs that were scheduled.

5.3.3 The Worker

The workers are responsible for running the test jobs. Not all workers are the same. They each have their own CodeWarrior versions installed and support some projects so that a test session that wants to run for a project doesn't have to wait for the test sessions running for other projects to end. Thus, a bottle neck may be avoided. Its workflow is described in Figure 17.

The worker polls the Job Queue waiting for tasks that he can run (new jobs with appropriate CodeWarrior versions and project names). Once a job is obtained, it updates its *status* to *running* and its *machine* to the worker's name and retrieves its SCF.

After parsing the SCF, based on the fields from the Job Queue, the worker determines the source files it needs to obtain from AccuRev from the appropriate stream.

The pre-processing script is ran from a command line shell using Python's *subprocess* module. The path from which the script should be called is specified by the user in the SCF. It is the shell's responsi-

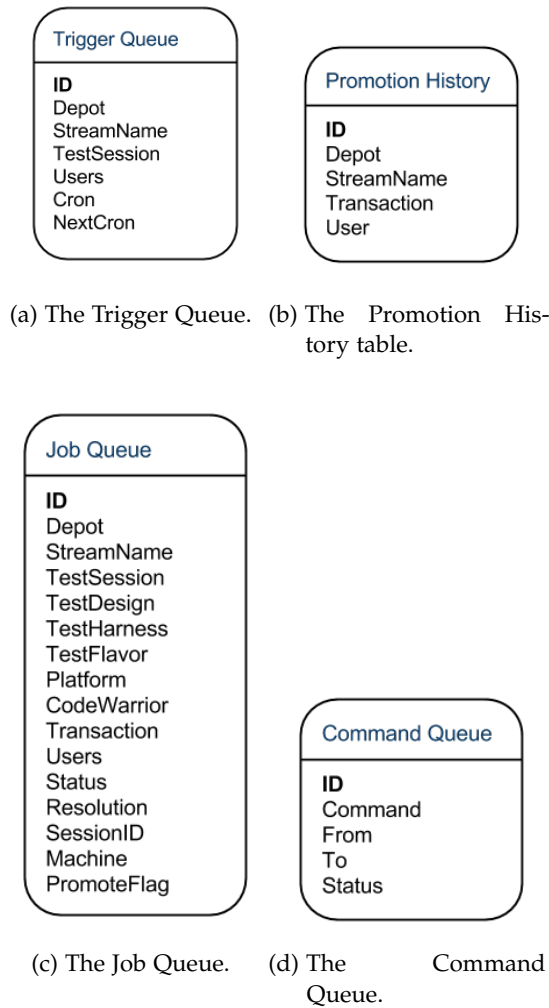


Figure 15: The ATF's database.

bility to determine with which interpreter to run the script. The ATF captures the script's output and logs it to a file.

Next, the test's dependencies are built. If there are components referenced in the test harness' flavor, their preprocessing macro definitions are updated using the *ecd.xe* CodeWarrior tool (see Listing 3). The same tool is then used to build the components with the targets specified in their flavors.

The test itself is built following the same steps as the components.

In order to run the actual test, a board from the board farm must be reserved so that no other users will intervene while the test is executing. This is achieved by first obtaining a list of usable boards and then attempting to reserve a random one from the list until an available one is secured. The list of potential boards is created by parsing the output of the *lars list boards* and the *lars info* commands and searching for the specified platform and the user defined hardware tags.

After reserving a board, the test CodeWarrior project must be aware of the board's configurations in order to run on it. Thus, the project's XML [RSE](#) configuration files must be parsed and the correct board settings must be introduced (see [Listing 1](#)). All the relevant board informations are obtained from the Lars database and replaced in the XML files using regular expressions.

Once the CodeWarrior test project is aware of the board it has to use, it also has to be able to read the test input. The *cwide.exe* can accept a TCL debugger script but it can not pass it arguments. In order to overcome this limitation, we introduced a TCL Debugger Shell wrapper responsible for passing the appropriate arguments to the debugger script. As described in [Figure 16](#), the ATF will take the following steps in starting the debug process:

- writing the path to the test script, the path from where to all the test script and the path to the test case in a text file on disk
- calling the [IDE](#) by passing it the wrapper as argument
- the wrapper reads the paths from the input text file and calls the test script from the specified path with the test case as argument using the TCL *source* command covered by a *catch* statement
- when the test script ends its execution, the wrapper saves its result and writes it to a separate text file on disk, then quits the [IDE](#)
- if errors occur, the wrapper catches them in the *catch* statement and writes them to the result file
- the framework then proceeds to read the test's result or errors from the result file on disk.

As specified in [Section 3.2](#), some errors are bound to occur during the test's execution. The ATF has a list of known errors that it tries to overcome by relaunching the test when they occur. Furthermore, if CodeWarrior happens to freeze for any reason, the ATF works as a watch dog, force killing the [IDE](#) after a timeout.

The test's resolution is set by the output of the test script. Pass/fail messages are expected.

The post-processing script is ran in an identical manner to the pre-processing one.

If the pre/post-processing scripts return errors the job is aborted by setting its *status* to abort and its resolution to *error*.

If the build fails, or the test script fails, the job is finished with the resolution *fail*. If the test script throws errors, the job is finished with the resolution *error*. Otherwise, the job's resolution is set to *pass*.

In the end, after setting the test's resolution, all the logs generated by the pre and post processing script, the build processes and the

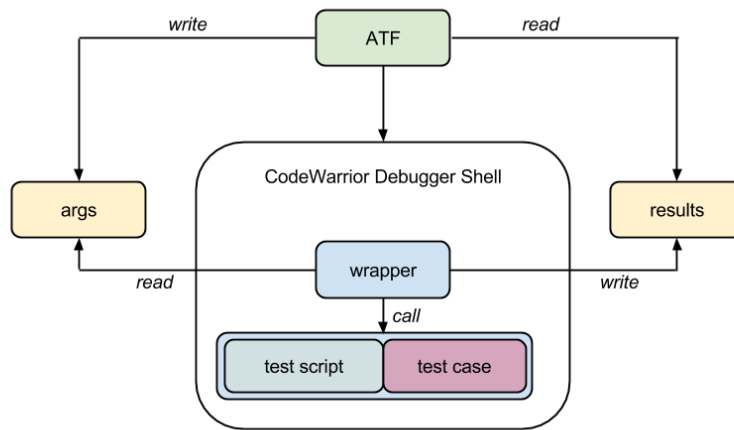


Figure 16: The CodeWarrior Debugger Shell wrapper.

test script, along with the report files specified by the user in [SCF](#), are gathered in an archive and copied to a shared location (the Linux machine running the CCS Mediator). The archive also contains a *pickled* Python object describing the job's configuration and resolution. The archive's name contains the job's ID and the session's ID for it to be easier to distinguish from the others.

Finally, the worker queries the Job Queue to see if all the jobs from the session have finished their execution. If so, it sends a command to the result aggregator through the Command Queue to let him know he needs to send the final report for the specific session. The command is `report <session number>`. The structure of the Command Queue is described in [Figure 15d](#).

The pickle Python module is used for serializing and de-serializing Python objects to and from text files by converting them to and from byte streams.

5.3.4 The Result Aggregator

The aggregator polls the Command Queue for commands addresses to him (whose `to` field contains his name).

Once such a report command is found, it parses it to obtain the session ID it needs to address. It then retrieves all the result archives from the shared location, unzips them and zips them back into a single one. This final archive is put back to the shared location for the users to retrieve them.

The pickled job objects are de-serialized from file and consulted in order to determine the session's final resolution and to compose the mail report. The aggregator will also promote the files from the specified transaction of the session passed and has the promotion flag activated (in the case of [GS](#) generated sessions).

The mail report contains, for each job, the following information:

- overall resolution

- build status (for each dependency)
- build warnings count
- run status
- run duration
- the name of the board that was used
- the name of the worker on which the job ran
- details extracted from the [SCF](#) such as the test design, the test harness, the test flavor, the platform and the CodeWarrior version

Finally, the report is sent to the users.

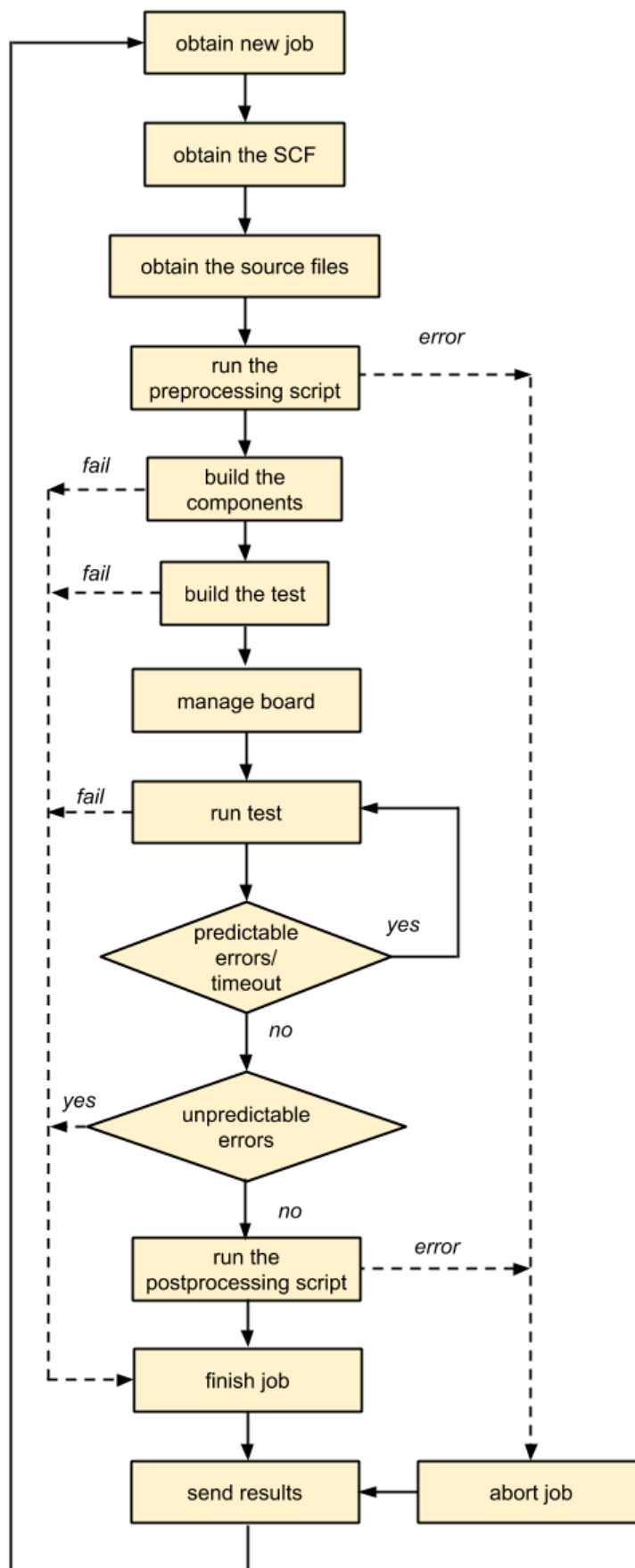


Figure 17: Worker block diagram.

RESULTS

In this chapter we will illustrate some of the [ATF](#)'s accomplishments, as well as some of its shortcomings.

Comparing it to its predecessor's, the framework's value is indisputable due to the [SCF](#) which grants its users the freedom to define their test sessions as divers as required.

We can analyze the framework's performance by studying the time it takes the developers to run various tests.

The baseband engineers working on implementing the [LTE](#) standard on B4860 [QDS](#) platforms develop multiple modules in parallel. Each module requires to be validated by its own tests. Below we briefly describe some of these modules:

- Module 1: tests the Downlink Control Channel ([DLCCCH](#)) module in charge of transmitting the base station's¹ control messages to the User Equipment ([UE](#))².
- Module 2: tests the Physical Uplink Shared Channel ([PUSCH](#)) module responsible for carrying the [UE](#)'s data to the base station³.
- Module 3: tests the Physical Downlink Shared Channel ([PDSCH](#)) module, the base station's main data bearing channel, in a non synchronized manner
- Module 4: tests the synchronized [PDSCH](#) module
- Regression Test: an extensive test session that covers all of the project's modules

As mentioned in [Section 5.2](#), the [SCF](#) allows the baseband engineers to divide a test case into multiple ones so that they can be run in parallel as [ATF](#) jobs. In [Table 1](#) we compare the previously described modules in order to emphasize the time difference between executing them by a developer and running them with the framework. Seven workers were used and a set of 14 boards. For a better visualisation of the same data, see [Figure 18](#).

In order to have each job independent from the others, one of the framework's shortcomings is that the workers need to obtain the source code and rebuild the projects each time. Thus, for short tests, the time it take for a worker to retrieve the source files from AccuRev,

¹ A base station is a transceiver that connects mobile phones to the telephone network

² An [UE](#) is an end-user device used for communication, such as a mobile phone

³ http://www.eetimes.com/document.asp?doc_id=1278199

Session	Manual Time	Job Count	ATF Time
Module 1	30m	2	17m
Module 2	1h	4	30m
Module 3	3h	10	1h
Module 4	4h 30m	10	1h 15m
Regression Test	18h	63	3h 30m

Table 1: Time to run a test session comparison.

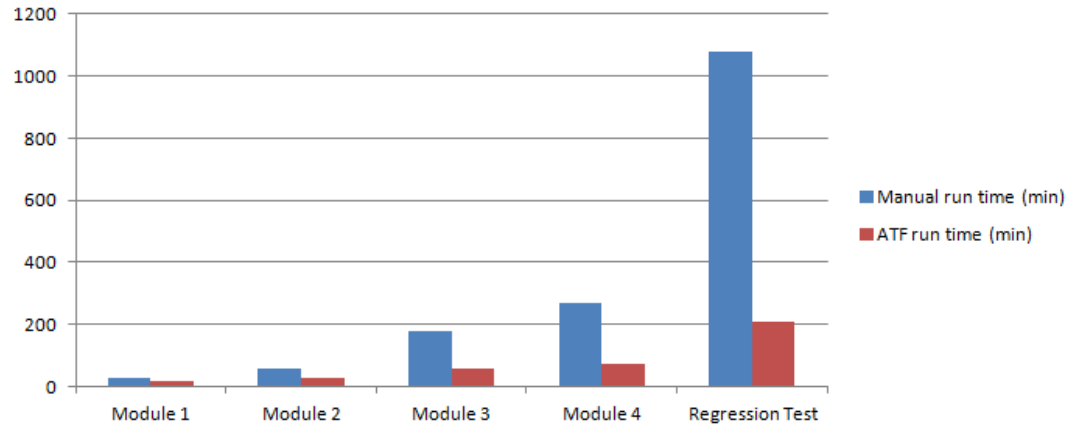


Figure 18: Time to run a test session comparison.

build the project and run it will take more time than it takes the project's developer to debug it by hand. This is a small impediment and, when such jobs make part of grand test sessions, their delay is unnoticeable.

FUTURE WORK

The ATF is a fairly complex environment but many features can still be added. In this chapter we will describe a couple of these possible additions.

7.1 COMMUNICATING WITH THE WORKERS

Test sessions may be launched by mistake or source files may be forgotten. In such scenarios, the developers may want to abort the launched test session and trigger a new one.

There currently is no direct method of aborting a job. The user need to contact the framework's administrator and ask him to restart the workers running a certain session.

The current method of stopping a worker is accessing the Virtual Machine on which it runs and killing it by hand. Furthermore, if the worker is executing a job when it is killed, the board it uses remains reserved in Lars and the job remains in a *running* state in the Job Queue.

It is desired for the users to be able to abort a running session using the ATF's client. This would be implemented using the Command Queue by sending messages to the workers in a similar manner as sending report requests to the Result Aggregator.

7.2 RUNNING A TEST SESSION FROM AN ARCHIVE

Before a project release, the developers strip the AccuRev stream of all non-essential files and obtain an archive with the files that will be delivered to their clients. The release manager may want to run a final set of validation tests on this archive before its delivery.

This task would required the framework to be capable of retrieving source files from an archive from a shared location as well as from AccuRev streams.

CONCLUSION

Testing is a major part of any application's development process. When practices such as [CI](#) are used, complex test sessions need to be run fairly often. Thus, an automated testing tool can facilitate a project's testing phase.

When it comes to baseband embedded projects, a hardware environment setup must be conducted before the testing the product. Due to this cause, the projects can not be supported by existing automated testing applications.

In this paper we described the design of the [ATF 1.0](#), a distributed automated testing framework aimed at baseband projects, and listed its limitations.

Next, we introduced the [ATF 2.0](#), a more easily configurable and user friendly version than its predecessor. We illustrated its components, we outlined its workflow and, finally, we reported its performances.

All in all, the framework's implementation was a success. At this time, the [ATF](#) is used by baseband engineers and its development will continue.

BIBLIOGRAPHY

- [1] Martin Fowler. Continuous Integration. 2006.
- [2] British Computer Society Specialist Interest Group in Software Testing. *BS 7925 - 2 Standard for Software Component Testing*. British Standards Institute, 1997.
- [3] Kshirasagar Naik and Priyadarshi Tripathy. *Software Testing and Quality Assurance*. John Wiley & Sons, Inc., 2008.