CHALMERS

Testing a Software Block with QuickCheck

Master of Science Thesis in Computer Science and Engineering

JIA WANG
SHYUN SHYUN YEOH

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2009

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Testing a Software Block with QuickCheck

JIA. WANG,
SHYUN SHYUN. YEOH,

© JIA. WANG, June 2009.
© SHYUN SHYUN. YEOH, June 2009.

Examiner: PATRIK. JANSSON

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2009

Abstract

This thesis has delivered a prototype to prove that it is technically feasible to test a software block written in C++ with QuickCheck in Erlang. The prototype consists of two parts. The first part solves the message passing communication problem and presents a scheme to translate Erlang messages to C++ objects automatically and vice versa. The second and final part is a QuickCheck state machine implementation with automatically generated test case generators. The correctness of the system under test is established by verifying the expected output signals based on QuickCheck generated input signals.

Acknowledgments

We would like to express our greatest gratitude to the following individuals in making this thesis project possible and a success.

We would like to thank Roger Holmberg, Mike Williams and John Hughes for giving us the opportunity to perform this thesis work.

We would like to thank our examiner Patrik Jansson for his guidance and patience.

We would like to thank our supervisor, who happens to be one of the greatest snow boarders in the company history, for facilitating and helping us to complete the project.

We would like to thank Hans Svensson and John Hughes for their enormous assistance during the course of the project.

We would like to thank Tomas Johansson for sharing his work on how to connect C++ and Erlang.

We would like to thank Pers Karlsson and Lars Jonsson, for showing us the software block and test framework.

We would like to thank Thomas Arts for the QuickCheck course.

How to Read This Report ?

- Chapter 1 Introduction
  The first chapter presents the motivation, problem definition, goal, task and scope of this project.

- Chapter 2 Background
  Chapter two prepares the readers with sufficient background knowledge to follow all the topics of this project comfortably.

- Chapter 3 QuickCheck
  We dedicate a chapter to introduce QuickCheck to the readers.

- Chapter 4 Analysis
  This chapter documents the design of our solution.

- Chapter 5 Implementation
  All the implementation details are captured in this chapter. We present the interplay between all the related components by examples.

- Chapter 6 Test Results
  This chapter discusses a bug in the application found by QuickCheck.

- Chapter 7 Conclusion
  Here we conclude our thesis report by sharing the lessons we learned during the course of the project and future work proposals.

Terms & Abbreviations

- `CEM`, Cell Manager: A software block in the SCC subsystem. It handles cell capabilities. This is the SUT.
- `PBC`, Pray Before Compile. A non technical procedure to peace the god of compiler.
- `RAN`, Radio Access Network: The mobile telecommunication network between end users and the core network.
- `RBS`, Radio Base Station: A component in RAN. It handles radio connections from mobile devices users.
- `RBSOS`, Radio Base Station Operating System: The main application running on and controlling the RBS.
- `RNC`, Radio Network Controller: A component in RAN. It controls RBS's and manages their connections to the core network.
- `RoseRT`, Rational Rose RealTime: A `UML` based development tool used to develop complex, realtime, concurrent system.
- `SCC`, Sector & Cell Control: A subsystem of RBSOS. It controls radio hardware and manages carriers.
- `SUT`, System Under Test.
- `UML`, Unified Modeling Language: A standardized general-purpose modeling language for software development. It include graphical notation to visually represent a system at an abstract level[8].
- `XMI`, `XML` Metadata Interchange: Object Management Group (OMG) standard for exchanging metadata information via `XML`[9].
- `XML`, eXtensible Markup Language: General-purpose specification for creating custom markup language[7].

Contents

Abstract iii

Acknowledgments iv

How to Read This Paper ? v

Terms vi

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Purpose 1
1.2 Problem Definition 2
1.3 Goal 2
1.4 Task 2
1.5 Scope 2
1.6 Deliverable 3

2 Background 4

2.1 Mobile Telecommunication Networks 4
2.2 RBSOS 6
2.3 Sector & Cell Control, SCC 7
2.4 Cells 8

2.5 Rational Rose RealTime, RoseRT 8
2.6 Scripted Regression Testing Today 9
2.7 Software Testing 10

3 QuickCheck 14
3.1 Quviq QuickCheck 14
3.2 Properties 15
3.3 Generators 16
3.4 QuickCheck State Machine 18

4 Analysis 21
4.1 Where to Start and End 21
4.2 Communication 22
4.3 How to Use QuickCheck 24
4.4 Signal Specifications 25
4.5 Usability 26
4.6 Pointers and Private Attributes 26

5 Implementation 29
5.1 Communication 29
5.2 Marshaling Code Generation 33
5.3 QuickCheck 35
5.4 General Measure 39

6 Test Results 41
6.1 Error Found 41
6.2 Test Conclusion 42

7 Conclusion 43
7.1 Lessons Learned 43
7.2 Future Work 43

Bibliography 45

List of Tables

5.1 The gen_server callback functions 32
5.2 Callback functions for QuickCheck statem behavior module . . 36
5.3 Other modules used to run QuickCheck 36
5.4 The gen_server callback functions 39

List of Figures

2.1 Radio Access Network 5
2.2 Subsystems of a RBSOS 6
2.3 CEM as a block in SCC 7
2.4 Geographical view of cells 8
2.5 Two connected capsules 9

3.1 A complete cycle of a QuickCheck state machine test case . . 19

4.1 Current Deployment 21
4.2 Finished prototype 22
4.3 Just another Erlang node 23
4.4 A and its public clone ghost_A 28

5.1 Connecting Nodes 30
5.2 Sequence diagram showing how erlAdapterC works 31
5.3 How to generate marshaling code 33
5.4 How we generate QuickCheck generators 37
5.5 Erlang side architecture 39

Chapter 1

Introduction

This chapter outlines the purpose, problem definition as well as the goal of this thesis project.

1.1 Purpose

This project is commissioned by a world-leading supplier in telecommunications. Telecommunication applications are some of the most complex applications ever produced. While applications with increasing complexity promise more functionality, such complexity often comes with a side effect of creating more room for errors.

Therefore, in order to develop complex applications with less errors and in a shorter time, our commissioner is constantly searching for faster and more cost efficient software testing solutions. QuickCheck, a relatively new testing tool developed in Erlang by a company called Quviq, presents itself as a potential candidate that fits the profile. Some of the areas in which QuickCheck excels over current test tools are automatic test cases generator and the ability to locate the heart of any errors found by shrinking the test case that causes an application to fail to a minimal one. With these promising features, our commissioner would like to know whether it is technically feasible to test software blocks with Quviq QuickCheck.

1.2 Problem Definition

The problem we will answer: Can QuickCheck be used to test software on the block level?
We break this problem in two:

1. Since QuickCheck is implemented in Erlang and the software block in `C++`, can Erlang communicate effectively with the software block using a message passing mechanism?
2. If so, by using the result from question `1`, can we test the software block using QuickCheck?

1.3 Goal

The goal of this project is to prove that it is technically possible to a test software block with QuickCheck and we support our claim by developing a working prototype.

1.4 Task

1. Build an infrastructure allowing Erlang and `C++` to communicate using message passing mechanism.
2. Implement QuickCheck to test the software block on top of the result from task `1`.

1.5 Scope

This prototype is not a finished product directly usable in a software development environment. It is limited to the scope defined in this section and served as a proof of concept.

1. The software block is SccCem.
2. SccCem is implemented in `C++`.

3. We test the SUT using four signals, namely, `createCell`, `deleteCell`, `setupCell` and `releaseCell`.

4. We will not evaluate non-technical aspects of QuickCheck, e.g. time, cost etc.

1.6 Deliverable

Beside this written report, all implementations during the course of this project, namely the prototype, will be delivered internally to our commissioner upon the completion of this thesis project. Due to confidentiality issues, we obscure all the implementation details without loss of academic values to this report.

Chapter 2

Background

This chapter provides the background information needed to read through this report. It is our goal to make this report as self-contained as possible.

We offer a non-technical view starting from the telecommunication networks for mobile networks down to the location of the SUT in such networks. Subsequently, we introduce the Rational Rose RealTime since the SUT and part of our implementation are implemented using this tool. We will then end this chapter with topics in software testing apart from QuickCheck which will be presented in a dedicated chapter.

2.1 Mobile Telecommunication Networks

Up until the year 2008, there were approximately 3.4 billion mobile phones[2], three times as many as there were fixed line telephones[1], globally. To be functional, each and every one of these mobile phones needs to connect to the mobile telecommunication networks.These networks are gateways, figuratively, to the connected world be it fixed telephones, other mobile networks or the Internet. A typical and simplified version of a mobile telecommunication network is shown in figure 2.1 and it is called RAN, radio access network. End users access these networks with mobile devices, typically cell phones, capable of transmitting radio signals in a predefined frequency range.

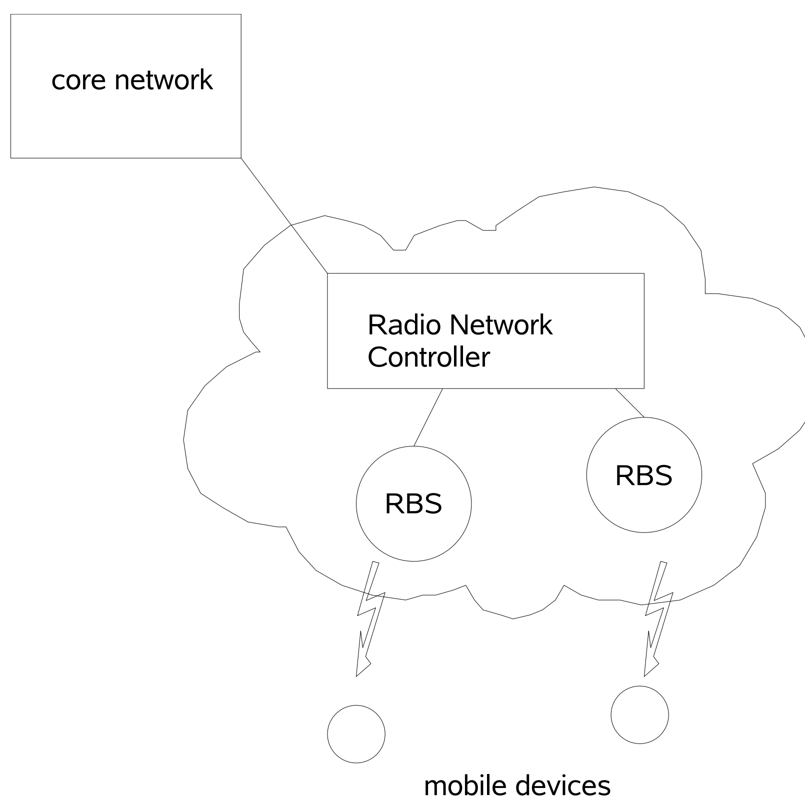

Figure 2.1: Radio Access Network

The RAN is the first network a mobile user connects to before any further communication with the rest of the world is possible. It enables the connectivity between the users equipments (mobile phones) and the core network. The core network represents the rest of the connected world, namely fixed telephones, other mobile phones and more recently the Internet. There are several components inside a RAN, they are the Radio Network Controller and the RBS's, Radio Base Stations. Radio base stations handles the radio connections directly from mobile users. RNC, in turn, controls the RBS's and manages their connections to the core network. The software block we are testing lives in the operating system of an RBS as explained in the subsequent section.

2.2 RBSOS

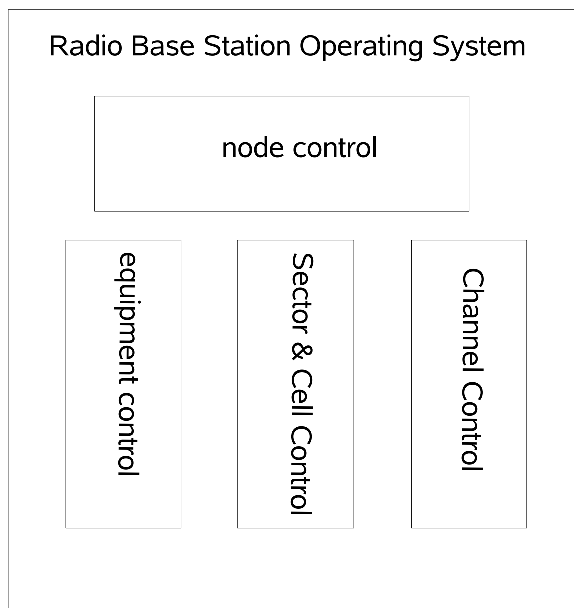

Figure 2.2: Subsystems of a RBSOS

RBSOS is the operating system running in a RBS. Like the more familiar operating system for your computer, it manages resources and interfaces RBS to other components in a network. The RBSOS consists of four major subsystems as shown in figure 2.2. The system under test is a software block resides in the SCC, Sector & Cell Control subsystem.

RBSOS subsystems, figure 2.2:

- NC, Node Control handles the interface to the RNC using NBAP protocal externally.
- EC, Equipment Control configures and supervises hardware.
- SCC, Sector & Cell Control controls radio hardware and manages carriers.
- CHC, Channel Control manages common and dedicated channels.

2.3 Sector & Cell Control, SCC

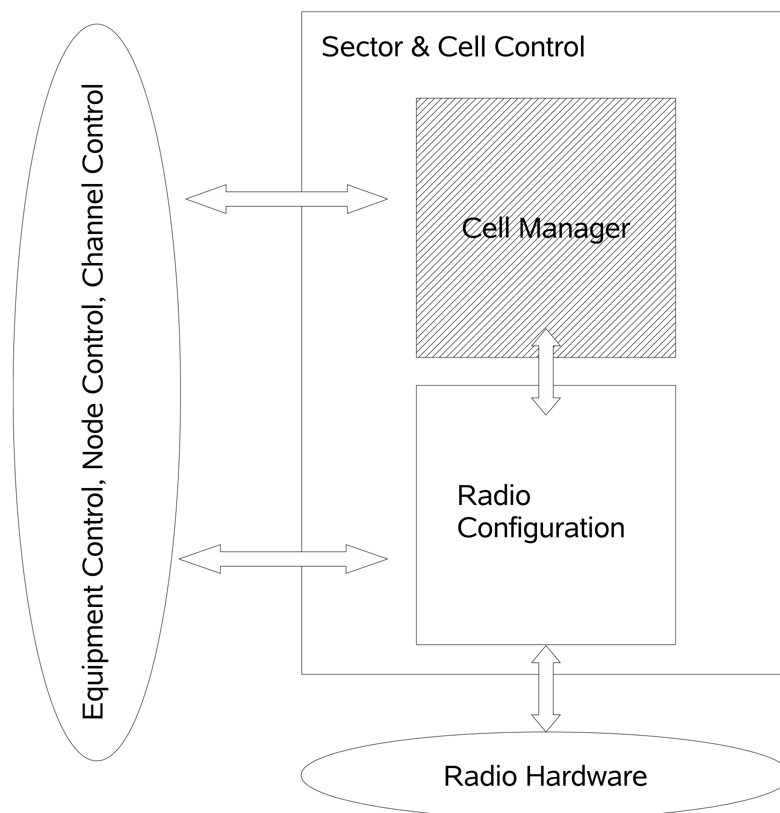

Figure 2.3: CEM as a block in SCC

The software block, CEM, assigned by the commissioner is part of the SCC subsystem as shown in figure 2.3.

- RC, Radio Configuration sets up and supervises sectors and carriers.
- CEM, Cell Managers handles cell capabilities. It is to this block that we will send signals to create, delete, setup and release cells and observe the corresponding returned signals.

2.4 Cells

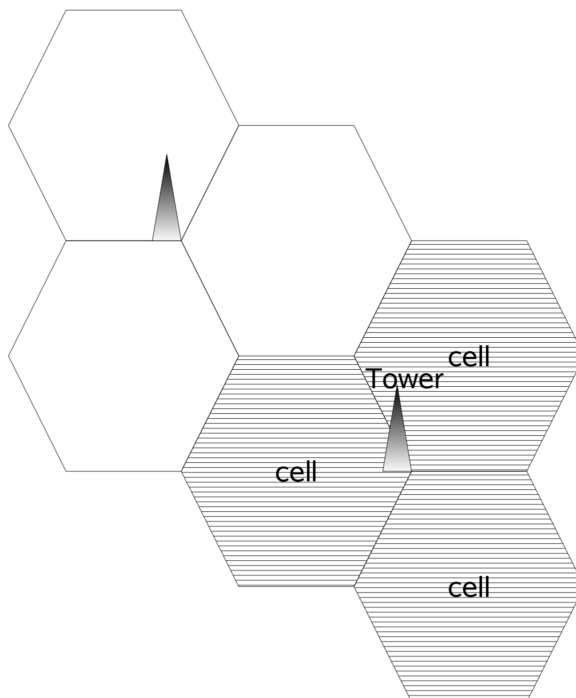

Figure 2.4: Geographical view of cells

We test signals which manipulate cells. Figure 2.4 illustrates geographically three cells (shaded) of different radio frequencies covered by a radio tower. In this project, we will send a sequence of commands manipulating certain aspects of cells to the SUT and verify the returned signals. Some of the signals are illegal like an attempt to delete a non-existing cell.

2.5 Rational Rose RealTime, RoseRT

The software block is modeled with a tool called RoseRT[6]. RoseRT is a development tool used to develop software in UML, the Unified Modeling Language. On top of the strength of UML to express high-level system properties visually, RoseRT adds some realtime notation to UML. One of the features this added realtime notation provides is the actor model concurrency. In actor model concurrency, each concurrent entity is self-contained (no shared memory) and can only communicate with other entities by passing messages. The connection between `RoseRT` and `C++` is that every time

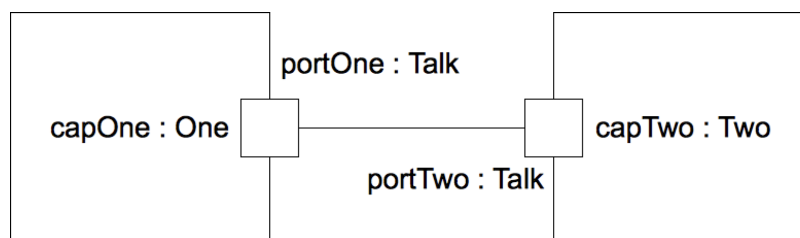

Figure 2.5: Two connected capsules

we try to build a `RoseRT` model, `RoseRT` will first generate the `C++` code of that model and build the `C++` code.
Some of the realtime modeling constructs are:

- Capsule: capsules are light weight concurrent objects of a special form of class defined in `RoseRT`. They are highly encapsulated and only communicate with other capsules through a message-based interface called ports. Erlang users should be able to recognize the similarities to Erlang process since the concurrency in both languages are modeled, to a different degree, after the actor model.
- Ports: ports are objects used by capsules to send messages to other capsules.
- Protocols: Every port is associated with a protocol. A protocol defines what messages can be sent and received from a port.

Figure 2.5 shows a simple example of `RoseRT` model built using capsules, ports and protocol. In this example, there are two communicating capsules, `capOne` of type `One` and `capTwo` of `Two` are connected to each other using the `Talk` protocol through port `portOne` and port `portTwo` respectively.

2.6 Scripted Regression Testing Today

We identify some of the disadvantages of the scripted regression tests used today in contrast to QuickCheck.

The first problem is that the telecommunication software is so complex that the traditional script based testing approach is unable to cover the execution paths effectively. For a particular module, there are dozens of protocols defined by the specification and each protocol contains a group of corresponding signals to achieve different goals. Applying some automated

testing tools makes it easier to repeat the whole test suite automatically, but the system testers still have to create a great many test cases manually. For example, the cell creation operation is a basic step during an RBS operation and each cell is created by a create cell signal containing seven arguments. It is impossible to create test cases to cover all the combinations of possible values for all the arguments. Consequently, only a few typical values are used in current test cases. Moreover, the problem will increase exponentially as more cells get involved.

The second problem is that even if a test case leading the system to an error state is found after 1000 signals are sent, it might be very difficult for the programmers to recognize the root of the error let alone correct it unless a small enough case leading to the same problem can be found. Although there are some available guidelines to shrink the failure cases to smaller ones, it is still costly to create these smaller test cases manually and repeat the test again and again. During this process, a lot of time is wasted in creating these similar failing test cases.

Another serious problem is the lack of flexibility to adapt static test cases to cover system changes and this will decrease the reusability. As communication technology developes, the product will update accordingly. The system was designed to be flexible. However, this is not true for static test cases since each test case is a fixed group of data. To update the test code, system testers have to locate all the changes in the new version requirement and change the test cases accordingly. During this time, the working test cases have to be rewritten and validated one more time. When finding a mismatch between expected results and actual ones, the testers have to make sure the test code is correct and the real errors are indeed error in the system code. This is why the testers can not reuse existing test code to test a new version of the software.

2.7 Software Testing

Software testing[4] is a process in which a program is executed with the intention of finding errors in the system under test. Testing is all about finding the yet undiscovered errors. Contrary to common beliefs that successful testings are those that discover no errors in the system under test, a test should only be considered successful if it does discover errors of some kind. From this perspective, testing should not be geared towards the direction of establishing the correctness or compliance of a piece of software to its specifications, but to find errors. Compliance and correctness are only the conclusion we

draw after failing to find any errors under the given resources. However, we will conform with the convention that a test is successful if no errors are found since QuickCheck, our testing tool, adopts this convention.

2.7.1 Principles of Software Testings

There exists no perfect software. We will try to convince readers of the non existence of perfect software by arguing that it is practically impossible to establish that a software is perfect. Since there is no perfect software, software testing is a not process to establish that a piece of software is correct but to find the maximum number of errors with the least resources.

2.7.2 Black Box Testing

Black box testing is testing without knowledge of the internal workings of the SUT. During the test, a set of input data is fed into the SUT and the corresponding outputs are compared against the expected outputs, according to specifications. Black box testing, in some ways, models the end users' experience and expectations of using a software application. End users, when using an application, with no concerns over how it works, would make inputs according to their needs and expect a reasonable outcome. Black box testing assumes the role of end user but rigorously tests all the outcomes against the specifications.

2.7.3 Exhaustive black box testing

One of the most direct, probably the only, ways of establishing full confidence in a software application is to test it exhaustively. In a black box setting, the testers would feed all possible and allowed input combinations into the system under test and observe the results. This approach is totally reasonable and, if succeeds, would instill full confidence. However the catch is computational and economical implausibility of exhaustively testing an application with all possible inputs. Say we have a program that would sort ascendingly 10 integers ranging from 1 to 100. This is a common program in our everyday lives and features in the first chapter of the introduction to algorithms course. In order to exhaustively test it, we have $100^{10} = 10^{20}$ different combinations of inputs to be tested on. Even if we are able to test 1000 cases per second it still requires approximately $3 \cdot 10^{10}$ years to complete the test. And this is not even a commercially valuable program, for example a program that

sorts 1000 files in a directory in a file system according to file size, where the volume of data is usually much higher.

2.7.4 White Box Testing

On the opposite end is white box testing. White box testing takes full advantage of the knowledge of the internal workings of the system under test. Instead of just feeding inputs and observing outputs, we use the logical structure of the system to design our test cases.

2.7.5 Exhaustive White Box Testing

However, the advantage of being able to exploit the internal structure of a system does not contribute to efficient exhaustive testing. An exhaustive white box testing requires the testers to traverse all logical paths of the system under test. In a 30-statements program with a finite loop where the loop body consists of several if branches, we need to traverse all possible combinations of paths from the starting point until end as many times as the finite loop. Mathematically, it is similarly unachievable as in the case of exhaustive black box test.

2.7.6 Grey-Box Testing

A hybrid of white and black box testing is the grey box testing. Test cases are derived or enhanced with the knowledge of the internal workings of the system under test but the tests executed as if the system under test is a black box. This is the strategy we are adopting for our testing. We would identify some of the signals needed for testing from the UML model (white box) and feed the signals to the system under test and test the returned signals (black-box).

2.7.7 Regression Testing

Regression testing means running the same test before and after changes are made to the system under test. This test is important to ensure that changes made do not break the original code. New tests however have to be devised to ensure the correctness of the new code.

2.7.8 Block Testing

A block is a package of classes that work as a collective unit to provide a well-defined functionality. Our software block of interest is the cell manager block in the sector & cell control subsystem. In terms of RoseRT, it is a package of communicating capsules that is responsible for cell management; namely the creation, deletion, setup and release of cells just to name a few. Therefore, we classify block testing as a grey box testing of this particular block of software. We use QuickCheck to generate signals that can be fed to the block and check the returned signals.

Chapter 3

QuickCheck

This chapter gives an introduction to basic QuickCheck usage and the QuickCheck state machine which will be used to implement our tests.

3.1 Quviq QuickCheck

QuickCheck[5] is a property based testing tool originally written in Haskell and has now been implemented in other major languages. The version we are using is Quviq QuickCheck, a version of QuickCheck implemented in Erlang produced by a company called Quviq. The tool runs tests based on properties, not manually written test cases. QuickCheck frees up the intense labor of writing individual test cases, which are often insufficient in capturing the essence of the specifications, allowing testers to focus on a larger picture of writing properties in a concise way that the system under test ought to fulfill.

This chapter should provide enough background information so that the readers are able to follow the QuickCheck related discussions in subsequent chapters.

There are two features that set QuickCheck apart.

1. QuickCheck randomly[1] generates a large number of test cases using generators, and runs tests against the given properties.

[1]not in the mathematical sense since QuickCheck would start with generating small values

2. Should the test case fail, QuickCheck will not just return the found counter example but a (locally) minimal counter example, this feature is called shrinking.

3.2 Properties

Properties are logical (first order logic) descriptions, for example a logical description that says for all empty list, the length of the list is zero. QuickCheck represents properties as functions.

```
prop_lists_doubleReverse()->
  ?FORALL(L, list(int()),
          L == lists:reverse(lists:reverse(L))).
```

Code 1: QuickCheck property to test lists:reverse twice

This is a property, Code 1, that tests the reverse function provided in the standard Erlang lists module. What it describes is that for all lists L of arbitrary length and of random integer elements, the result of reversing it twice is the same as the original list L. Imagine the effort needed to test this function by manually writing test cases. But QuickCheck is able to randomly generate test cases (one hundred of them by default) and check that the list that has been reversed twice is still the same as the original one.

```
3> eqc:quickcheck(test:prop_lists_doubleReverse()).
..................................................
..................................................
OK, passed 100 tests
true
```

Code 2: successful test

This, Code 2, is the result of running QuickCheck on the previous property. the prefix `test` is just the module name in which `prop_lists_doubleReverse` is defined. It takes no time to finish the 100 tests. We should not read anything more from the test result than what is stated. It says that the property has passed 100 tests, but by no means a guarantee that `lists:reverse` a list

twice will always return the original list. What we might get after running a large number of these tests is that we have more confidence in the behavior of lists:reverse.

Next, we will show what a failing test looks like. For example assume we misread the lists documentation and come to believe that `lists:usort` is just a sorting function (it is more than that), then it should hold that the length of the list before and after `lists:usort` are the same. We have:

```
prop_lists_usort()->
  ?FORALL(L, list(int()),
          length(L) == length(lists:usort(L))).
```

Code 3: QuickCheck property to test lists:usort

we get:

```
11> eqc:quickcheck(test:prop_lists_usort()).
.......................Failed! After 24 tests.
[-5,-5,3]
Shrinking.(1 times)
[-5,-5]
false
```

Code 4: QuickCheck property

This shows that `prop_lists_usort()` does not behave as expected, i.e. the length of list before and after lists:usort are different, and the test failed after 24 test cases. The counterexample is [-5,-5,3], and the minimal counterexample after shrinking, is `[-5,-5]`, i.e. `length(lists:usort([-5,-5]))` is not equal to `length(list:usort([-5,-5]))`. Further investigation reveals that `lists:usort` sorts and removes all duplicates from the list, namely lists:usort([-5,-5,3]) = [-5,3].

3.3 Generators

A crucial component in random test case generation is the QuickCheck generators (with built-in shrinking behavior). QuickCheck provides a number of

basic generators for example:

- `int()`: a random integer generator.
- `bool()`: a random boolean generator.
- `list(int())`: generates integer list of arbitrary length.

By combining the generators, we can define any test data generators we need. Here, Code 5, we define an Erlang record that represents a signal named `connectCall` which is a `C++` object of class call with an integer attribute and a boolean attribute.

```
-record(signal,{name, type, values})

signal()->
#signal{
        name = connectCall,
        type = call,
        values = [int(), bool()]
       }.
```

Code 5: Signal Generator

Some possible generated values for generator `signal()` in Code 5 are listed in Code 6.

```
{signal,connectCall,call,[9,false]}
{signal,connectCall,call,[6,true]}
{signal,connectCall,call,[-11,false]}
{signal,connectCall,call,[6,true]}
```

Code 6: possible generated values for signal()

A example scenario to put all these together:
For the test environment, we have connected to another Erlang node acting as an echo server. The echo server is implemented in `C++` and will instantiate (according to protocol) the Erlang tuples it receives into `C++` objects, the

process is called unmarshaling, and translate them back to Erlang tuples, marshaling, and send them back to the Erlang node from where the tuples are received. A property that would test the correctness of the marshaling and unmarshaling code is:

```
prop_signals()->
  ?FORALL(S, signal(),
          S == send_receive(S)).
```

Code 7: QuickCheck property to test marshaling and unmarshaling code

Referring to Code 7, `signal()` is a generator for generating all the signals as defined in the marshaling and unmarshaling protocol. And the `send_receive(S)` function will send the signal `S` to the echo server and return the received the echoed signal.

3.4 QuickCheck State Machine

Unlike the `lists:usort` example shown in previous section where we were testing a pure function, the SUT is a stateful machine with side effects, so we need to model it with a state machine in order to run a better test. For our test, we implement QuickCheck state machine to model the SUT. By implementing a state machine, we are able to generate more meaningful commands instead of a random sequence of commands which are not very helpful in testing a stateful machine.

For instance if we try to QuickCheck a lock state machine with two stages `open` and `locked`, at some stage, we might want to generate an unlock command only when the machine is in the `locked` state which correspond to how we actually use a lock, instead of depending on randomness to occasionally stumble upon such coincidence.

QuickCheck state machine offers the ability to:

- initialize the model with predefined state data.
- model a stateful machine by passing state data around.
- generate a sequence of commands.

- use `precondition` to decide what commands may be added to the sequence
- use `postcondition` to verify the state after a command is invoked.

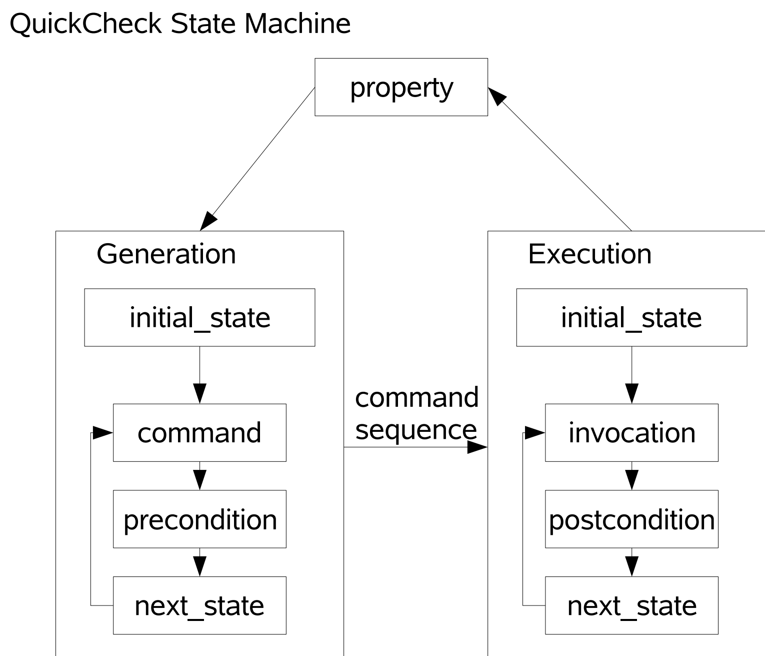

Figure 3.1: A complete cycle of a QuickCheck state machine test case

There are two stages involved when running the QuickCheck state machine, the generation stage and the execution stage. Figure 3.1 illustrates how a test case is generated and executed. It is as easy to run one such test case with QuickCheck as it is one thousand.

The cycle is started by the QuickCheck property function. Then QuickCheck will generate a sequence of commands which will be invoked in the execution stage. One main distinction between generation stage and execution stage is that, no command is actually invoked during the generation stage, or in QuickCheck terminology, those commands are symbolic calls. Symbolic calls are just a symbolic counterpart of commands that are actually invoked. When we discuss the state machine, we are referring to the QuickCheck state machine (model), not the state machine of the SUT.

Generation

1. `initial_state`: initialize the state machine to a predefined state. This guarantees that every command generation starts from a common ground.

2. `command`: generate command sequence. The length of the sequence can be a predetermined or a more random one provided by the QuickCheck `size` parameter.

3. `preconditon`: filter the generated command from `command` (in 2). The function `precondition` is called to determine if a generated command is actually what we want based on state data. We have no control over what commands are actually generated by `command` once they are defined, however we can decide whether a generated command will be added to the command sequence.

4. `next_state`: update the generation stage state machine.

5. `2,3,4` constitute a cycle of generating one valid (passed the `precondition` test) command which will be added to the sequence. This cycle will loop until the a certain length, as mentioned in `2`, of the sequence is reached.

6. The generated sequence (symbolic calls) will be passed to the execution stage where they will be actually invoked.

Execution

1. `initial_state`: similar to `initial_state` in generation stage.

2. invocation: invoke one command from the command sequence.

3. `postcondition`: verify that the result of actually invoking a command corresponds to our model of the SUT.

4. next_state: update the execution stage state machine.

5. `2,3,4` constitute a cycle of invoking a command from the command sequence. This cycle will loop until the command sequence is empty or a failed test case is found.

6. return to `property`

Chapter 4

Analysis

This chapter presents a guided tour of the problem solving strategy and design decisions. All the implementation details are presented in chapter 5.

4.1 Where to Start and End

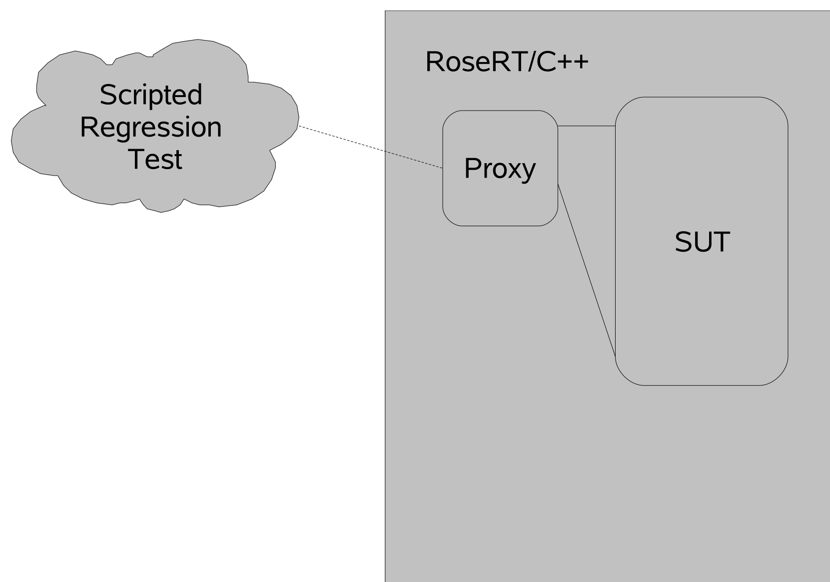

Figure 4.1: Current Deployment

Figure 4.1 depicts what we are assigned to by the commissioner and the current test environment. The ultimate goal is to develop a prototype as in

figure 4.2. This figure will be further explained in chapter 5 and the subsequent sections in this chapter, the main idea to get from these two figures are that we have replaced the scripted test framework with QuickCheck.

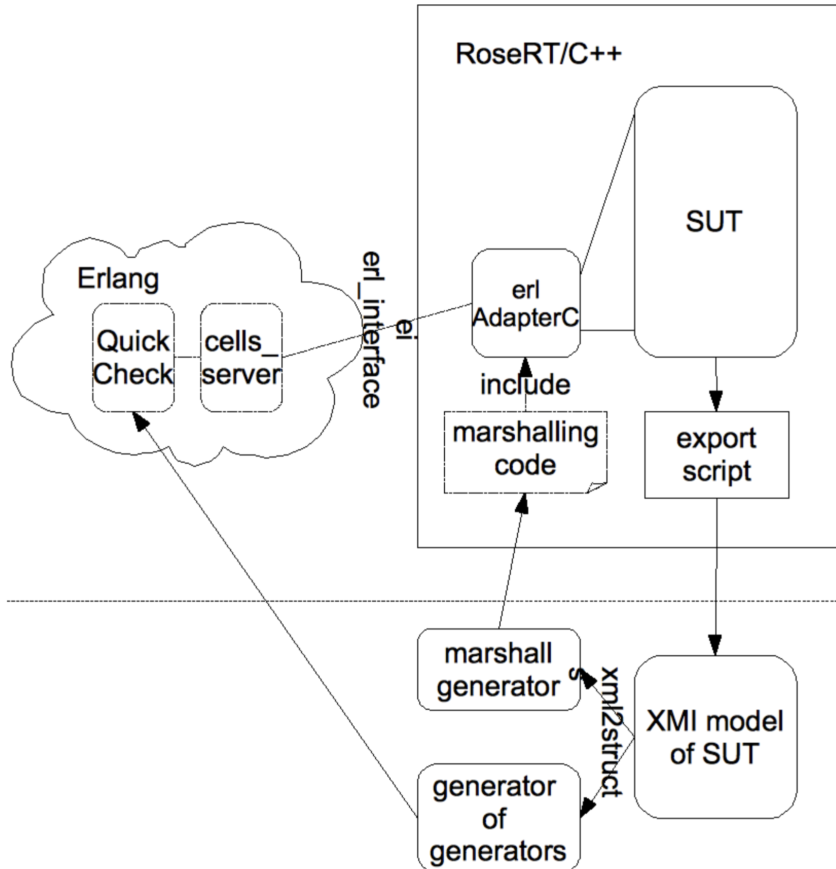

Figure 4.2: Finished prototype

4.2 Communication

4.2.1 Communicating Nodes

The first problem is to develop a framework to send messages from Erlang to the SUT and vice versa. We use the `ei` and `erl_interface` libraries, they are C interface libraries for communication with Erlang. These two are standard libraries in the Erlang distribution and support the following:

1. manipulation of data represented as Erlang data types.

2. conversion of data between C and Erlang formats.
3. encoding and decoding of Erlang data types for transmission and storage.
4. communication between C nodes and Erlang processes.

From Erlang point of view, we have wrapped the SUT as a Erlang node, called `cnode`, as in figure 4.3. Now we can view the whole problem as an Erlang problem. This is an ideal scenario because QuickCheck can then be used to test the SUT as any other Erlang application.

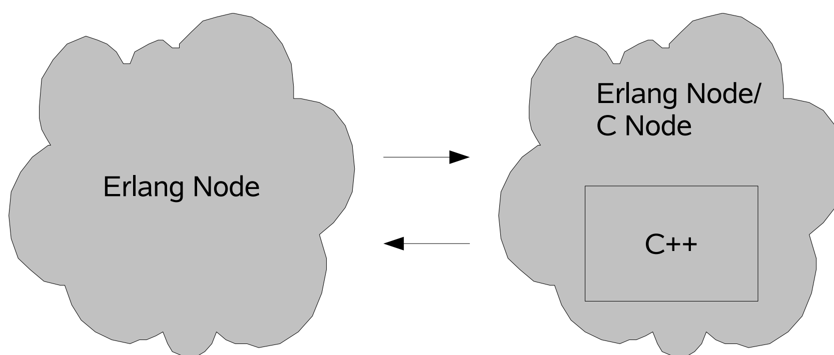

Figure 4.3: Just another Erlang node

4.2.2 Marshaling Code Generation

We call the translation of data structures from `C++` to `Erlang` marshaling and unmarshaling the opposite. However, we will use the term marshaling for general discussion purpose.

The result from the previous subsection 4.2.1 allows Erlang and the SUT to hear each others voices, without understanding the content of the conversation. In order to understand each other, we need a protocol for the translation of data structures between `C++` and Erlang. We need a scheme, marshaling, to translate an Erlang representation of `C++` data structures to `C++` and back.

The reasons we want to have automatic marshaling code generation are:

- the data needed for marshaling code generations are well-structured `XMI`. This makes it easier to run filters or extract information that we need.

- the number of classes involved are overwhelming.
- less human coding errors.
- works for other software blocks.

4.3 How to Use QuickCheck

QuickCheck is used to generate a large number of signal sequences using the QuickCheck State Machine and verify the returned signals. This is achieved by implementing the QuickCheck state machine call back module.

4.3.1 Properties, Preconditions and Postconditions

We write QuickCheck properties, preconditions and postconditions based on the signal specifications that will be discussed in the coming section 4.4

4.3.2 State Machine

We implement the QuickCheck state machine as a callback module to the QuickCheck `statem` behavioral module.

The reason is that we want to have some degree of control over the generated sequences of signals instead of total randomness. By using a state machine, we are able to model the SUT and generate signals based on our state machine model.

4.3.3 Generators

Writing QuickCheck generators manually is a tedious job because the number of data classes involved are overwhelming. For example, one of the signals that we implement, the `setupCell` signal, contains 20 attributes some are which are arrays and other data classes. Furthermore, we want to automate as many processes of using this prototype as possible to facilitate its probable incorporation into development environment. Therefore, instead of writing individual generators, we implement a generator of QuickCheck generators to automatically generate all the QuickCheck generators needed from the SUT model.

4.4 Signal Specifications

We run our tests by sending sequences of QuickCheck generated input signals to the SUT and verify the corresponding output signals from the SUT.

Even after we have decided to take the signal verification approach, we still have to decide on the signals to be verified. We pick the following signals because they represent the main functionality of the software block and they cover all of the more complex `C++` data structures defined in the SUT, namely enumerations, private attributes, pointers and arrays. Even though not all data classes defined in the SUT are tested, the not tested signals are structurally similar to one of the data structures that is covered by one of these four signals. A valid signal (signal that fulfills its condition in the following list) triggers a corresponding confirmation signal from the SUT, otherwise a rejection signal.

1. `createCell`:
   function: create a cell.
   condition: parameters different from any current cells[1].

2. `deleteCell`:
   function: delete a cell.
   condition: delete a current cell that has not been set up.

3. `setupCell`:
   function: set up a cell.
   condition: set up a current cell that has not been set up

4. `releaseCell`:
   function: release a cell
   condition: release a cell that has already been set up.

Examples of valid sequences of these signals are:

- `createCell X` → `setupCell X` → `releaseCell X` → `deleteCell X`.
  SUT output signals sequence: `createCell X` confirmation → `setupCell X` confirmation → `releaseCell X` confirmation → `deleteCell X` confirmation.

- `createCell X` → `deleteCell X`.
  SUT output signals sequence: `createCell X` confirmation → `deleteCell X` confirmation.

[1] cells that have been created and not been deleted

Examples of invalid sequences of these signals is:

- `createCell X` → `deleteCell Y`.
  SUT output signals sequence: `createCell X` confirmation → `deleteCell Y` rejection.
  Reason: cell `Y` has not been created yet.

- `setupCell X`.
  SUT output signals sequence: `setupCell X` rejection.
  Reason: cell `X` has not been created yet.

From the above examples, the signals that we verify are the sequences of output signals (dependent on our input signals) from the SUT.

4.5 Usability

We want to make our prototype reusable with minimal modifications. The usability considerations result in the following:

1. Separate module and capsule to hide the connection implementation for `Erlang` and `RoseRT` respectively.

2. Automatic marshaling code generation.

3. Generator of QuickCheck generators.

4.6 Pointers and Private Attributes

4.6.1 Pointers

When we first encounter a pointer, we look at it as an integer since a pointer is a memory location. But soon we realize it is not reasonable for QuickCheck to generate random integers for pointers since we have no control over what object really is at that random memory location. So instead of generating random memory address for a pointer, we generate random object of type similar to the base type of that pointer.

In this example, `X` is the randomly generated object of type `ClassA` (or NULL for null pointer) represented by the Erlang tuple for pointer `ptr`. So

when this message is received at the `C++` side, we will first unmarshal the object `X`, then assign its memory location to `ptr`.

```
Erlang
{pointer, ptr, ClassA, X}

C++
ClassA* ptr=&unmarshal_ClassA(X);


or


Erlang
{pointer, ptr, ClassA, NULL}

C++
ClassA* ptr = NULL;
```

Code 8: pointer representation in Erlang and its corresponding unmarshaling code in C++

The marshaling of a pointer is the opposite process.

4.6.2 Private Attributes

Some of the data classes we encounter contain private attributes. This poses a problem to our approach of directly assigning unmarshaled values to all attributes. For the sake of completeness, we want QuickCheck to have the freedom of generating values for both private and public attributes. To overcome this barrier, for every data class with private attributes, we define a ghost class with the same attributes with one exception that all the ghost attributes are public. The idea is that we could then manipulate the pointer of the ghost class and therefore manipulate the private attributes.

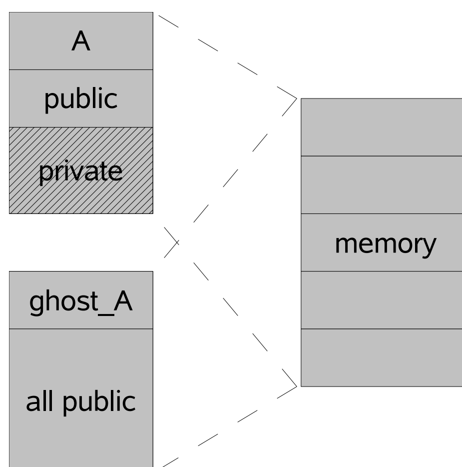

Figure 4.4: A and its public clone ghost_A

On the same region of memory, from `A` perspective, there are certain memory locations that cannot be accessed directly, but to `ghost_A`, every memory location is directly accessible(figure 4.4).

We summarize our strategy as follows:

1. for each class `A` with private attributes, we define a clone of `A` called `ghost_A` with one major difference that all attributes in `ghost_A` are public.

2. `ghost_A` is an exact public copy of `A` solely to allow us to manipulate the private attributes of `A` through `ghost_A`.

3. cast and assign the pointer, `ptr_ghost_A`, of object `ghost_A` to a pointer to the object type `A` that we wish to manipulate.

4. now we manipulate the all attributes of object type `A` through `ptr_ghost_A` "legally" (though breaking all the rules of abstraction barriers) since all attributes in `ghost_A` are public.

Chapter 5

Implementation

This chapter contains a description of the implementation of this project.

5.1 Communication

A gen_server is a generic server Erlang behavior module. This module provides a generic client-server server on top of which a server of specific functionality are built. This generic server takes care of all the underlying communication details allowing us to focus on the function of the server. The module that defines the functionality of the server, for instance as a HTTP web server or chat server etc., is the call back module to the generic server behavior module.

We have implemented a gen_server callback module, cells_server on the Erlang side and a capsule erlAdapterC in RoseRT to handle the communication using the erl_interface and ei libraries as shown(shaded) in figure 5.1.

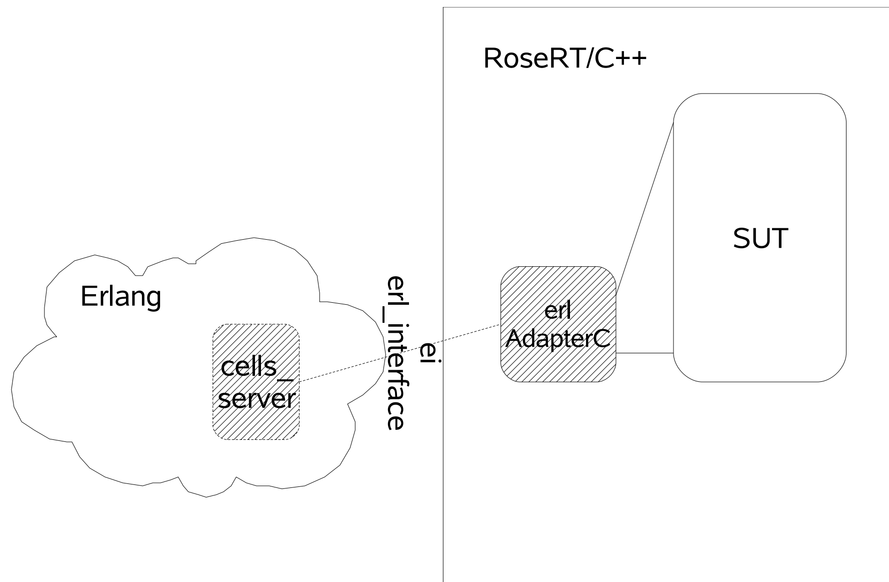

Figure 5.1: Connecting Nodes

The erlAdapterC in figure 5.1 is a wrapper around the erl_interface and ei libraries API handling connections set up, sending and receiving of messages and name registration on the RoseRT side. On the other hand, the cells_server is the equivalent in the Erlang side. It implements the gen_server callback functions instead of library functions.

5.1.1 erlAdapterC on the RoseRT side

There is another capsule, erlMessageReceiverC, inside erlAdapterC, and its main job is to listen to messages from Erlang. The two capsules are packaged as a RoseRT package called rrt2erl to ease the process of porting it to other software blocks.

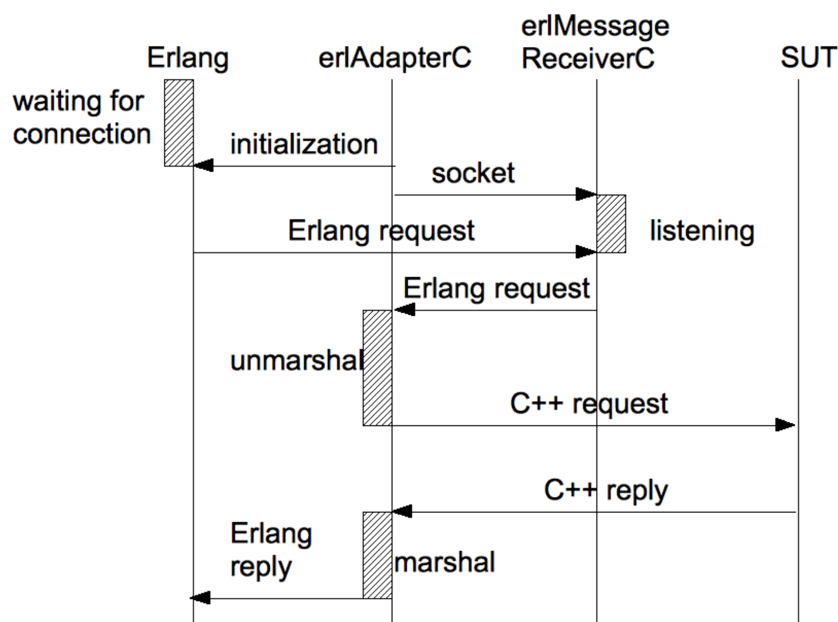

Figure 5.2: Sequence diagram showing how erlAdapterC works

erlAdapterC works as follows(figure 5.2):

- erlAdapterC initializes communication with Erlang.
- It then sends the socket and delegates the listening job to erlMessageReceiverC.
- If any messages are sent from Erlang, erlMessageReceiverC will send the message to erlAdapterC.
- Once a message is received from erlMessageReceiverC, erlAdapterC will unmarshal the message and send it to the SUT.
- As soon as the SUT replies, erlAdapterC will then marshal the reply and send it to Erlang.

5.1.2 cells_server on the Erlang side

The cells_server module is a callback module to Erlang gen_server generic server.
cells_server implements the following callback functions:

Callback functions	Summary
start_link	start the server
init	initialize the internal state of the server and the SUT.
handle_call	handle specific requests to the server.

Table 5.1: The `gen_server` callback functions

The following is a list of features of `cells_server`:

- the `init` function performs a series of message exchanges with the SUT to initialize the SUT. One of the signals for instance, is an array of parameters representing a license of that cell. License restricts the types of cells that can be created.
- the `init` function waits for the initialization message and the process ID, `Pid` of `erlAdapterC` and stores the `Pid` as internal state.
- Distribute messages between the SUT and QuickCheck.

5.2 Marshaling Code Generation

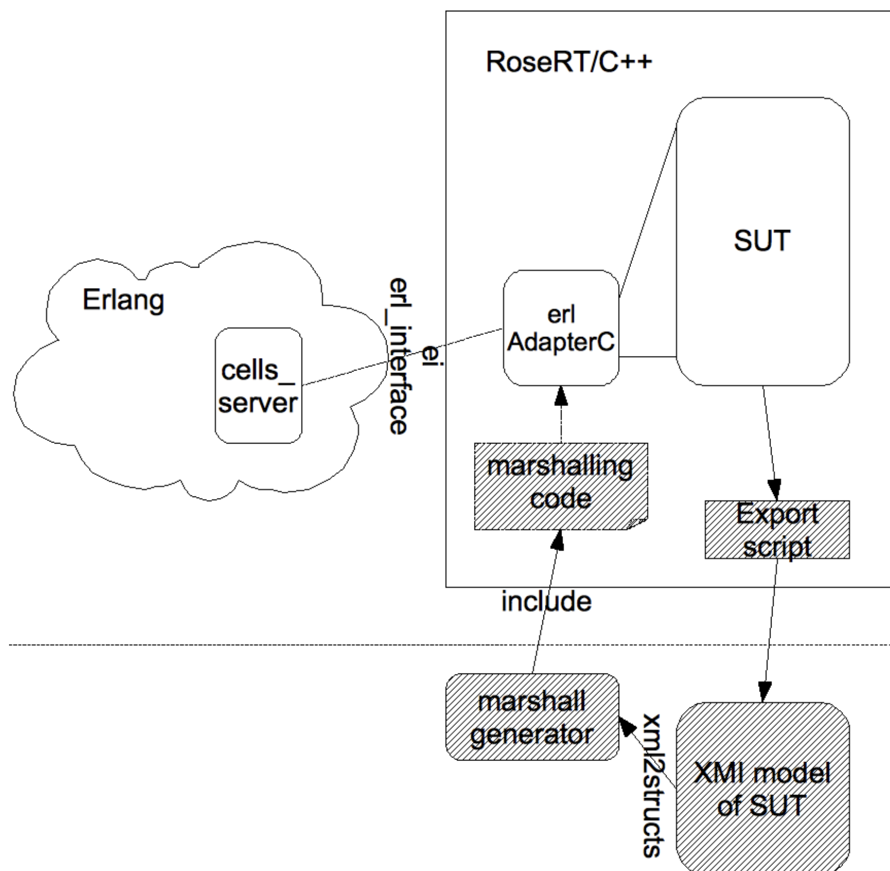

Figure 5.3: How to generate marshaling code

The code generation is implemented in Erlang and we obtain all the class, signal, port and protocol definitions needed from the `RoseRT` model as shown (shaded) in figure 5.3 and in the following steps:

1. export script: Export the `RoseRT` model to an `XMI` model, the script to do this job is readily available.
2. `xml2structs`: Process the exported `XMI` model and extract all the related classes into Erlang list of tuples (these tuples are the representation of `C++` classes, protocols, and ports).
3. `marshal_generator`: generate the marshaling code from the list of tuples.

4. Include the generated and static marshaling code into the SUT (the static marshaling code is the marshaling for basic data types, e.g. `int`, `float` etc.).

For a concrete example, we will walk through an example of marshaling code generation and the process of unmarshaling. For simplicity, we only consider one class, namely:

```
class MyClass{
   public:
   int n;
   float x;
 }
```

Unmarshaling code generation

1. from the `XMI` of the model[1], `xml2structs` exports a list of Erlang tuples representing `C++` classes. `[{MyClass, [{int, n}, {float, x}]}]` is the Erlang list (of one element) output by `xml2structs`.

2. From the list from 1, `marshal_generator` generates the following unmarshaling function:

```
void unmarshal_MyClass(MyClass* Tgt, ETERM *Erl){
  // the first element in Erl is class name, MyClass
  // the data is stored starting from the 2nd element,
  // erl_element(2,Erl) and erl_element(3,Erl).

  // attribute n is an integer
  unmarshal_int(Tgt->n, erl_element(2,Erl));
  // attribute x is a float
  unmarshal_float(Tgt->x, erl_element(3,Erl));
  }
```

Code 9: the unmarshaling function for `MyClass`

[1] the actual exported `XMI` model is approximately 10MB

ETERM, namely Erlang term, is a type defined in erl_interface library to express Erlang data structures in C, in our case it holds the received Erlang tuple. The job of the marshaling code is to transform this ETERM object into a C++ MyClass object defined in the SUT.

Unmarshaling

1. When an Erlang tuple {MyClass,1,2.0} is received, unmarshal_MyClass is called. Erl is an ETERM pointer to the received tuple (Code 9).

2. The second element, which is an integer 1, is unmarshaled and assigned to Tgt→n. Similarly for the third element.

3. The outcome is that the pointer Tgt of type MyClass now points to a MyClass object in which $n = 1$ and $x = 2.0$.

The code to unmarshal basic data types, namely unmarshal_int and unmarshal_float, is not automatically generated but is provided in a static marshaling code module that will be included into all generated code to handle basic data types.

Currently, the code generation works for all C++ basic types, enumeration, pointer and array.

5.3 QuickCheck

This section describes the interplay among all the modules needed to run QuickCheck.

5.3.1 QuickCheck State Machine and Other Modules

The module cells_eqc is the main QuickCheck module. In it, we implement all the callback functions as required by the QuickCheck state machine as summarized in table 5.2.

Statem Callback	Summary
initial_state	initialize the state machine before generation and execution
command	generate a sequence of commands. This sequence is the test data
next_state	state transition function
precondition	the precondition to be met before a command is added to the sequence of generated command
postcondition	the postcondition to be met after command has been evaluated
prop_X	QuickCheck property

Table 5.2: Callback functions for QuickCheck statem behavior module

Beside `cells_eqc`, we need other modules to run QuickCheck in agreement with our modularized design. We delegate different jobs to specialized modules as listed in table 5.3 and figure 5.5.

Other Modules	Summary
gengen	the generator of QuickCheck generators
cells.hrl	Erlang records definition for all the classes that will be sent to or received from the SUT. These are the records that will be generated by QuickCheck
cells_generator	the QuickCheck generators generated by gengen. The scope of the generators is limited to those records defined in cells.hrl
cells_eqc	callback module for QuickCheck statem(the main module)
scccem_control	use system process to kill any unterminated instances of the SUT. There can only be one instance of the SUT running since the registered name used for communicating with Erlang is hard coded.
SccCem	the executable SUT

Table 5.3: Other modules used to run QuickCheck

We separate the records definition and generators from the main code into a header file, `cells.hrl`, and a normal module, `cells_generators.erl`,

respectively(figure 5.5). The advantage of doing this is that the header file can be included whenever needed.

The process of generating QuickCheck generators is shown in figure 5.4. This process is almost identical to the process for marshaling code generation.

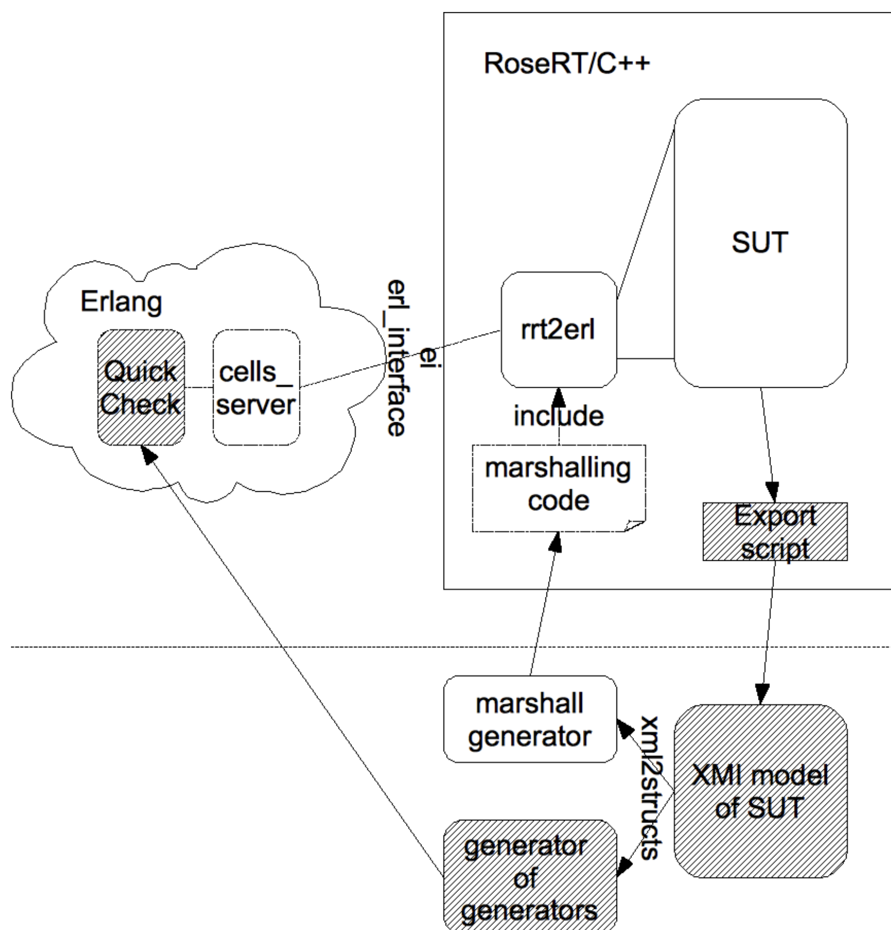

Figure 5.4: How we generate QuickCheck generators

Given an `XMI` file, all the classes definitions are available, thus it is convenient to extract all the class fields information from this file. For each class in `XMI` file, there should be a corresponding record in header file, then this transforming approach, from `XMI` file to an Erlang header file can be done easily.

The record generators are defined in a similar pattern. We just need to define a function named with each record and return a new record, giving each field an initial value. This value may be a basic type or composition

type. For a basic type, such as integer or bool, the QuickCheck's default generators can do this job. For a composition type, we can define another record generator recursively and finally, all the fields can be specified by basic values.

Since these two jobs follow the same pattern for we can simply define a generator of generators, that is a function to extract all the information from the `XMI` file and generate a file to contain all the record definitions and another one consists of various generators for each record. This is especially useful for those complex signals such as `setupCell`, which contains dozens of fields in total. The major advantage of this automation approach is that it not only can avoid errors introduced by hand writing, but this generator of generator is also available to any other blocks even without any change.

Given these modules separately, the core module `cells_eqc` becomes very concise. To test cells related operation, it only need to define a command generator for creating, deleting, setup and releasing cells, pre and post condition for each command and corresponding properties to verify these commands. All other stuff such as message generators, sending and receiving messages to other process, maintaining the pid of SccCem are extracted out in separate modules and this makes it easy to change different parts to apply to new testing block.

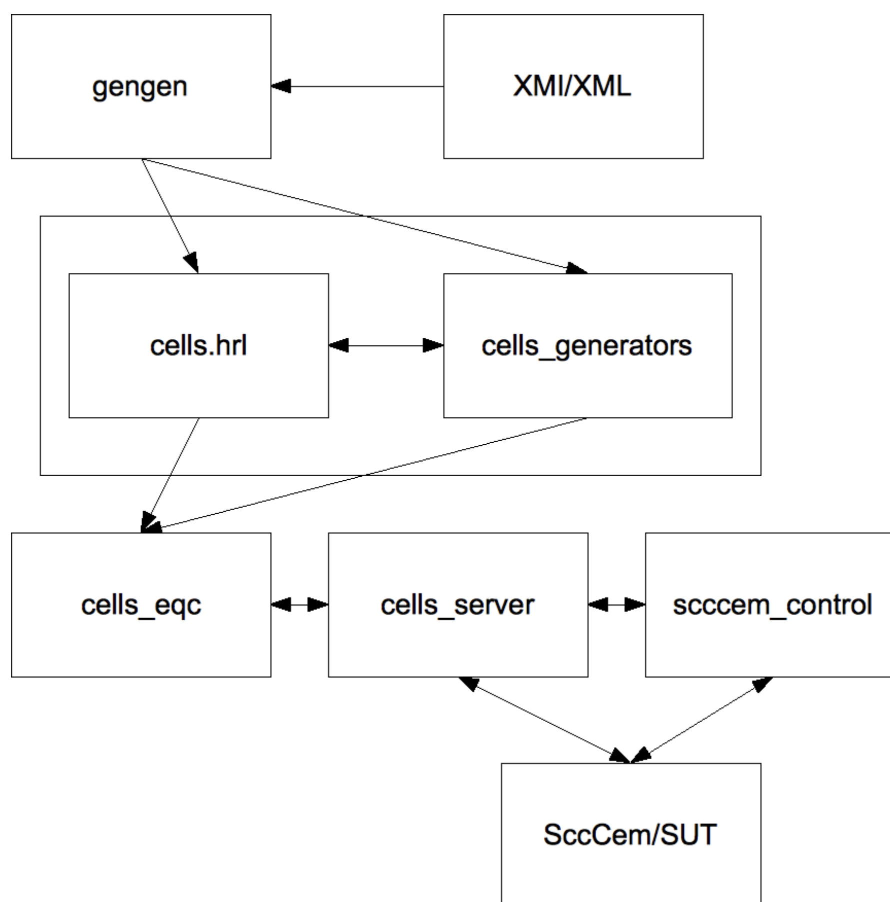

Figure 5.5: Erlang side architecture

5.4 General Measure

This section summarizes the codes and their sizes as a general measure.

Code	Lines of Code
cells_server	460
cells_eqc	190
gengen	130
marshall_generator	400
xml2structs	320
generated marshaling code	1400

Table 5.4: The `gen_server` callback functions

This project was carried out by the two authors of this report at a big telecommunication company. It is a 20-week project from 19 January 2009 to 1 June 2009. We spent an estimated 7 hours every working day during that period to work on this project. The codes are developed in pair with the occasional but highly appreciated help from John Hughes, Hans Svensson and Tomas Johansson. Besides the codes, 95% of this report is done during that period.

Chapter 6

Test Results

This chapter discusses the error we found in the SUT.

6.1 Error Found

Do notice that this error was found without explicitly writing any test cases. And the process of tracing the root of the fault has been enormously simplified by QuickCheck's ability to shrink failing test cases to the minimal case that triggers the same error. Apparently, this error has been missed by the currently employed static test cases.

We describe the failing test case here and analyze where the fault might be.

The test case that triggers this crash is a much larger one however QuickCheck shrinks it to the following minimal case.

1. create a valid cell with parameter `X`.
2. create cell confirmation is received.
3. create cell with the same parameter `X`.
4. as expected, this is not a valid creation since duplicated cells are not allowed. Create cell rejection is received.
5. However, cell created in 1 is a valid one and we wish to delete it now. A request to delete cell `X` is sent.
6. system crashes with memory error.

6.2 Test Conclusion

The error has been reported and fixed. The more encouraging fact is that QuickCheck has with relative ease revealed a bug in an application that has been used and tested for a long time.

Chapter 7

Conclusion

The test result presented in chapter 6 consolidates that our prototype works and that QuickCheck can indeed be used to test a `C++` software block.

7.1 Lessons Learned

- Do not continuously listen for incoming messages on a socket that is needed to send out other messages. We solved this problem with time-outs while listening.
- Make sure the buffer for sending and receiving messages between Erlang and the SUT is large or flexible enough for large messages. We solved this problems using a flexible buffer.
- We built our QuickCheck state machine (property, postcondition, precondition) on our understanding, which is far from complete, of the SUT. A deeper understanding of the SUT is essential in order to produce a better test.

7.2 Future Work

These are some future work proposals:

- Evaluate the values of this prototype from non-technical and technical aspects. The non-technical aspects being resources while the technical ones are code coverage etc. We summarize this proposal by formulating

a question: How good is this prototype compared with the existing ones?

- We discussed about export the SUT into a `XMI` model. From this exported model, we only extract class, signal, port and protocol definitions. The question is: Can we extract the state machine of the SUT from the `XMI` model and automatically generate a corresponding QuickCheck state machine? In our prototype, this is done manually.

- This prototype is not general enough. We are only confident that it will work on the particular software block on which we based our prototype using the four signals that we have implemented. So the next natural extension is to further generalize the prototype. One already known data type that fails our marshaling code generation is smart pointer.

- Model the SUT with the new QuickCheck finite state machine (this is different from the QuickCheck state machine that we have been discussing).

Bibliography

[1] Wolfram Alpha. land line - wolfram—alpha. http://www63.wolframalpha.com/input/?i=land+line, June 2009.

[2] Wolfram Alpha. mobile phones - wolfram—alpha. http://www.wolframalpha.com/input/?i=mobile+phones, June 2009.

[3] erlang.org. Erlang online documentation. http://erlang.org/doc/index.html, 2009.

[4] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons.

[5] Quviq. Quickcheck for erlang users, 2009.

[6] Rational the e-development company. Rational rose realtime modeling language guide, 2000.

[7] Wikipedia. extensible markup language. http://en.wikipedia.org/wiki/XML, May 2009.

[8] Wikipedia. Unified modeling language. http://en.wikipedia.org/wiki/Unified_Modeling_Language, May 2009.

[9] Wikipedia. XML metadata interchange. http://en.wikipedia.org/wiki/XML_Metadata_Interchange, May 2009.