Investigation and Modeling of Test Environment at DIPN/IMS

TOMAS MANNERSTEDT

KTH Computer Science and Communication

Master of Science Thesis
Stockholm, Sweden 2006
Investigation and Modeling of Test Environment at DIPN/IMS

TOMAS MANNERSTEDT

Master’s Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2006
Supervisor at CSC was Karl Meinke
Examiner was Stefan Arnborg

TRITA-CSC-E 2006:012
ISRN-KTH/CSC/E--06/012--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication
KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.csc.kth.se
Abstract

Investigation and Modeling of Test Environment at DIPN/IMS

This Master’s project examines the possibilities of an improved test environment at a unit for integration and functional testing of data switches within Ericsson. An improved test environment is achieved by supporting implementation of test cases as scripts. These scripts have to be able to communicate with network connected test instruments and test objects. The equipment communicates via an API either developed or planned to be developed by Ericsson or the vendor of the equipment. The engineers request the possibility to develop test suites in several programming languages, including Erlang and C. These languages are very different which make interaction between the languages difficult. Engineers that write in C does not want to learn Erlang and vice versa.

To solve the problem, a universal language for specifying interfaces on function level has been designed. For this language a parser has been developed which is used by the developer to automatically generate an API for both Erlang and C. The parser is developed in such a way so that support for other languages is easy to add. The language supports primitive types (strings and integers) as well as compound types (lists and tuples). In the interface the implementation language for functions is defined per function. The parser makes the function available from both Erlang and C.

Utredning och modellering av testmiljö vid DIPN/IMS


Acknowledgement

There are a number of people that I would like to thank. These are Joakim von Porat and Stig Jönsson for letting me do this Master’s project, Mike Allen and Ian Whyte for checking the spelling and grammar of the thesis, and all my other colleagues at XGA for great support and lots of fun chats.

Finally I would like to thank Professor Karl Meinke at KTH for great supervision, and my fiancée Ulrika Lindbäck for all her support during the project.
Contents

1 Background and Problem Description ......................................................................................................... 1
  1.1 Background ........................................................................................................................................... 1
  1.2 Goal .................................................................................................................................................... 1
  1.3 Testing at UDP/ET ............................................................................................................................... 2

2 Existing Tools .............................................................................................................................................. 6
  2.1 General ................................................................................................................................................ 6
  2.2 Test Tools .......................................................................................................................................... 6
  2.3 Languages ......................................................................................................................................... 8

3 Related Theory ......................................................................................................................................... 10
  3.1 Erlang ............................................................................................................................................... 10
  3.2 Software Architecture ...................................................................................................................... 11
  3.3 UML .................................................................................................................................................. 13
  3.4 CORBA .......................................................................................................................................... 14
  3.5 Formal Languages and Syntax Analysis .......................................................................................... 15

4 Solution ................................................................................................................................................... 19
  4.1 System Design ................................................................................................................................. 19
  4.2 Communication in Wrapper Layer .................................................................................................. 21
  4.3 Encapsulation of Wrapper Layer ...................................................................................................... 25

5 Conclusion .............................................................................................................................................. 34

6 Recommendations ................................................................................................................................... 35

Books ............................................................................................................................................................ 36
Ericsson Internal Documents ...................................................................................................................... 36
Internet ......................................................................................................................................................... 36

Appendix A: CORBA Client and Server ...................................................................................................... 1
  C Server .................................................................................................................................................. 1
  C Client .................................................................................................................................................. II

Appendix B: Router Tester and IS API Implementations ............................................................................. 1
  rt_impl.c ............................................................................................................................................... I
  RT-lib.tcl ............................................................................................................................................... I
  rct_impl.erl ........................................................................................................................................... II

Appendix C: Interface Parser Manual ......................................................................................................... 1
  Manual for Parser ................................................................................................................................. I
  Synopsis ................................................................................................................................................ I
  Description .......................................................................................................................................... I
  Options ................................................................................................................................................ I
  Interface Language .............................................................................................................................. I
  Example: ........................................................................................................................................... I
  Name Conventions ............................................................................................................................ II
  Limitations .......................................................................................................................................... II
  Author ............................................................................................................................................... II
1 Background and Problem Description

This chapter describes the background to this master’s project and the goals of the project.

1.1 Background

Development unit IP Network (DIPN) is responsible for development of the Ericsson All IP network. The IMS (IP Multimedia Systems) Gateway organisation, part of the DIPN, has the responsibility to define, develop and deploy gateway products supporting Ericsson’s IMS product portfolio. The UPD/ET unit, part of the DIPN/IMS Gateway organisation, is responsible for testing the IMS Gateway products.

The IMS Gateway products are tested in the UDP/ET lab. This lab contains test objects, tools for testing (traffic generators, traffic analyzers etc.) and terminals from which the test cases are executed.

Executing tests means to verify that the product fulfils the specifications regarding functionality, performance and robustness. Running a test case involves configuring the test object and test equipment, executing the test case and finally documenting the results. The setup of the test object and test equipment is currently made manually. This sometimes results in errors due to the human factor.

1.2 Goal

The goal of this project is to improve quality of the testing at the unit and increase the number of test cases being correctly performed. This will be achieved by investigating the current test environment and developing a model for improving the test environment.

1.2.1 Model Goals

There is a need to make test case scripting easier. The testers would like to easily develop test scripts that can be reused by other testers as well as for regression tests. The scripts need easy access to the configuration interfaces of the test objects and other test equipment like router testers etc.

The environment also needs real-time functionality. Both timers and timing functionality are needed.

There is a need for multiple interfaces to the environment. Some testers like to write their test scripts in Erlang, some prefer C. There is also a need to access different interfaces from the environment. Some test equipment is accessed via Tcl or sockets, and the test objects are often accessed via Erlang procedure calls.

The environment has to be easy to maintain and improve. This is important since the testers do not have the time nor the competence to maintain a complex software system. It is preferable if the different language interfaces can be automatically generated from an interface specification such as a C-header or an interface description.

This project does not attempt to make a program for automated tests, rather an environment for scripting in different languages.
It is preferable if the resulting model has different levels of abstraction so that test cases can be built on the optimal level of abstraction. This will result in a model where general test cases can be built quickly and simply. When the tester needs to write a more specific test case it can be constructed at a lower level of abstraction. Some tests cases can be implemented almost at the same level of abstraction, as the test specification while other test cases need to be written on a lower level of abstraction.

The results of this project will be a set of programs and/or functions to facilitate access to controlling different test equipment types, and configuring the system under test from a common interface. The aim is to limit the scope to cover the majority of the test suites and avoid an over complex implementation that may only be required for just a few specific test cases.

The keywords of the project are flexibility, simplicity, and modularity.

### 1.3 Testing at UDP/ET

*This section explains how testing is organized and performed at UDP/ET.*

#### 1.3.1 Test Organisation

The testing process at UDP/ET and Ericsson can be divided into several stages which are performed at different levels in the organization. These are in the order they are performed:

- Basic testing performed by design teams.
- SSIT: Sub System Integration Test.
- FT: Function Test.
- SIT: System Integration Test.
- EST: Early System Test.
- ST: System Test.

In theory the testing stages apply to the waterfall model. When the first test is finished the second begins and so on. However, the waterfall model is not practical since the time from idea to a product must be cut down. In reality the test process is carried out in parallel. This makes testing more difficult since all functionality can not be finished at all test stages, but it also cuts down the time from idea to finished product.

The unit in which this project is done is responsible for software and hardware SSIT and FT for the system.

#### 1.3.1.1 SSIT

SSIT is the testing where all the blocks from the sub units are put together and tested together. When performing SSIT it is important to verify that the connections and dependencies between the different subsystems work properly. Tests are not very focused on stability and not at all focused on performance testing. This is tested in other stages.

Since SSIT is done on a subset of a system all components are not present. This sometimes results in that the tests can’t be done as black-box\(^1\) tests.

At the responsible unit, SSIT is performed without a formal test specification. This is possible since all the testers are experienced and know what to test.

---

\(^1\) Testing without knowing anything about the underlying implementation.
Not much time is spent on SSIT since it is more important to deliver the subsystem to the next stage; SIT.

1.3.1.2 FT
In FT the tests are executed to verify that the sub systems fulfil the functional specification. FT is performed using a test specification.

1.3.1.3 SIT
SIT is performed by connecting the different sub systems together and testing the systems as a unit. During this stage the sub systems compatibility are verified.

From now on tests are supposed to be performed on fully equipped systems, however this is not always possible since much of the testing and development is done in parallel.

1.3.1.4 EST
During EST the maturity of the system is evaluated. When the system has matured it is passed on to ST. This is also a period for the ST team to learn about the product.

1.3.1.5 ST
In the ST stage the stability, robustness and performance of the system is tested.

1.3.2 Testing
This section describes how testing is done from a testers point of view. It describes how tests are being run on the current test object and current test tools.

1.3.2.1 Test Specification
The test specification is a document describing what to test in order to verify that the functions to be tested fulfil the requirements. The test specifications are written based on Function Specifications. A test specification contains several test cases, where each test case describes pre- and post-conditions for the test, and how to set them up.

In some cases all the tests cannot be executed due to lack of time. In these cases random samples are taken from the test cases and the testers prioritize tests on complex parts of the system. When selecting a subset of test cases to run, it is preferable if all the tests that are not performed are passed. This maximizes the number of errors found.

1.3.2.2 Setting up a test environment
Setting up a test environment can be a time consuming and painful task. Since the test object is newly developed it sometimes contains both hardware and software bugs.

To emphasize the complexity of setting up a test environment; the following interfaces are used to set up the suite of test tools:

- RouterTester: Setup is performed in a GUI via Remote Desktop, either provided by Microsoft or rdesktop².
- PacketStorm: Setup is performed either directly from the instrument or via a Java GUI started from a web browser.

² http://www.rdesktop.org/
• ANT20: Setup is performed either directly on the instrument or via an Erlang interface provided by Wipro.
• IS: Setup is performed via Erlang interface, SSH using Unix commands and a web interface.

Since so many different systems are involved in configuring the test environment it is not uncommon that something fails or is set up in the wrong way. Some instruments support saving a specific setup to a file, others do not. Therefore much of the work has to be repeated over and over again.

### 1.3.2.3 Running a test

The tests are performed by setting up the equipment according to the pre-conditions, executing the test and afterwards examining whether the post-conditions are fulfilled. The setup of the equipment is performed in different ways depending on the type of equipment. Different types of equipment are setup using different user interfaces.

In some cases tests are being run on test objects where some of the components are not yet finished or implemented. In those cases the missing components have to be simulated in some manner.

Sometimes it is possible to make automated tests. However, it is not always preferred since the non-automated tests result in a deeper knowledge of the systems. Automated tests are preferred for use during regression tests.

### 1.3.3 Practical test example

The goal of this test is to set up a link between Alice and Bob (see Figure 1). When the link is established traffic is sent and it is checked that no packets are lost.

In this case the SGC knows how to set up the link between Alice and Bob. Assume SGC is not yet implemented. This implies that the SGC has to be simulated in some way. The interface between SGC and the test object is Erlang based so the simplest way of simulating the SGC is to manually input the Erlang commands to set up the link between Alice and Bob.

Before the test can start we have to be sure that the environment is working properly. The steps to run the test will be:

1. Set up the Router Tester. This is done using Remote Desktop and a GUI-based application.
2. Run the Erlang commands in an Erlang shell to set up a connection between Alice and Bob.
3. Start traffic using the GUI for the RouterTester.
4. Wait for a while.
5. Stop traffic using the GUI for the RouterTester.
6. Remove the connection between Alice and Bob.
7. Check if any packets were lost using the GUI for the RouterTester.
Figure 1 Simplified schematic image of test setup.
2 Existing Tools

This section describes some of the existing tools that can be used to simplify and improve the test environment. The chapter is included in this thesis since the specification includes both investigation of existing tools and modelling of a new one.

2.1 General

Currently no specific test system is used at the test unit for testing. There are tools for handling test specifications, trouble reports and test statistics which are used, but no general way of making test suites. Since the specification of this project involves investigating existing test environments as well as modelling a new one, a few languages and a tool have been examined. Details on these will be explained here.

2.2 Test Tools

There are many systems for software testing available on the market. Many of these systems are developed for testing on a lower level than what is performed and needed in this case. The systems are often used for testing individual methods or classes rather than a full scale system with routers, gateways and switches; and therefore those systems are not capable of handling the task.

2.2.1 An existing test tool - LabVIEW

LabVIEW is a piece of software which can be used for testbench construction and is developed by National Instruments. Testbenches are constructed based on two views, the Front Panel and the Block diagram.

LabVIEW also includes an Expect-like (see 2.3.2) language to handle external interactive applications. This can be used to navigate setup menus or ssh connections to other systems.

2.2.1.1 Front Panel

The front panel (see Figure 2) is used to construct the user interface for the testbench. The GUI can contain buttons, input fields, graphs and different kinds of output fields.
2.2.1.2 Block diagram

The Block Diagram (see Figure 3) is used to construct the logical relations between the components of the testbench. The Block diagram can contain different types of signal generators, filters, triggers, logical statements and other components well known to a computer programmer.

Components are connected using wires in the block diagrams to construct programs.

Figure 2 Screenshot of the LabVIEW Front Panel.

Figure 3 Screenshot of the LabVIEW Block Diagram.
2.2.1.3 Conclusion
LabVIEW has many similarities with RAD\textsuperscript{3} programs. It is easy to build a simple application with GUI for some purposes. The application is shipped with many examples and templates to speed up development.

There are however a few disadvantages with LabVIEW. The simplicity of making programs makes the system somewhat inflexible. Many of the testbenches need to communicate with different languages and equipment and this might be a difficult task in LabVIEW.

Another problem with the application is the conservative users that dislike the click-and-draw technique to develop programs. LabVIEW has been an object of investigation at Ericsson. The program was met by mistrust by the engineers, mainly because of the user interface.

2.3 Languages
This section describes how an existing programming language can be used to write reusable test suites.

2.3.1 General
One approach to creating an easy to use test environment is to educate all the testers in some high level scripting language. This would make the test environment simple to use and understand.

There are several suitable languages and some of them are discussed here.

2.3.2 Expect
Expect\textsuperscript{4} is available both as a programming language and an API (libexpect) for C/C++, Perl and many other languages. Expect is a Tcl based language used to interact with interactive programs. An Expect program is built up like a dialog. The Expect-program says something and after that it expects an answer from the program it interacts with. This sequence in the programs has given the language its name. Depending on the answer the program takes different actions.

The Expect program can be programmed to switch between interactive and non-interactive mode. This can be used to let the user handle errors that occur during execution.

One benefit of using Expect as the language of choice is that scripts can be recorded directly by Expect using autoexpect. To do this the recorder is started, some action is performed and the recorder is stopped. The result of this is an executable Expect script, which can be modified to fit a more general purpose.

Another great advantage is that Expect is supported (as a library) by many languages. This allows testers to write their Expect-based test suites in almost any language. However, Erlang does not support Expect, which is the language of choice for many testers.

Expect is currently unknown to the testers at the unit but considered easy to learn by the community.

\textsuperscript{3} Rapid Application Development
\textsuperscript{4} Don Libes, 1995, Exploring Expect
2.3.3 Lua

Lua\(^5\) is a scripting language used for easy scripting. It is fairly simple to learn and powerful enough to write test suites. Lua would make a good choice since it is easy to learn.

The biggest disadvantage with Lua is that is currently unknown to the testers.

2.3.4 SWIG

SWIG\(^6\) is a software development tool for connecting programs written in C and C++ to different high-level programming languages like Perl, Python, Tcl/Tk, Ruby, C#, Common Lisp, Java, Modula-3, OCAML and Scheme.

SWIG can be used to connect a library (written in C/C++) to testbenches in different languages. Unfortunately SWIG does not support Erlang. Since Erlang support is important for the project, using SWIG is a bad idea.

One solution to the problem is to implement SWIG support for Erlang. Since time is limited this solution will not be further investigated.

2.3.5 Conclusion

Using a new programming language to implement functionality to control all test equipment and implement all testbenches in would make a good environment to work in. However, the testers and engineers at the unit are very conservative and do not like the idea of adding yet another programming language to their knowledge. The solution is much like avoiding the obstacle rather than dealing with it. Therefore this idea will not be further investigated.

\(^5\) [www.lua.org](http://www.lua.org)  
\(^6\) [www.swig.org](http://www.swig.org)
3 Related Theory

This chapter describes the theoretical background of the topics discussed in the final solution of the project.

3.1 Erlang

This section discusses Erlang\(^7\), its concept, origin and current use. Erlang is only discussed in a conceptual view since the syntax of the language lacks importance for the thesis. Erlang is discussed here since it is used at Ericsson for development and scripting.

For more information on Erlang, see Concurrent Programming in Erlang (Joe Armstrong, Robert Virding, Claes Wikström and Mike Williams, 1996).

3.1.1 General

Erlang is a programming language and runtime environment for general purpose. Its aims are to support concurrency, distribution and fault tolerance. Erlang is currently used in several large telecommunication systems from Ericsson and other vendors.

Erlang was developed at Ericsson Computer Science Lab during mid 1980’s. The unit was doing a project on a computer language which made it easier to program telecommunication programs. The people involved in the development were Joe Armstrong, Robert Virding and Mike Williams. Others joined later and added things to the distribution like OTP.

The syntax of Erlang has emerged from Prolog. Erlang started out as a modified version of Prolog. Some other syntax constructions have been borrowed from CSP and Eripascal (an extended version of Pascal).

Erlang is suitable for distributed, reliable, real-time and concurrent systems such as telecommunication systems (switching, converting protocols etc.), servers and internet application (IMAP-servers, HTTP-servers etc.) and database applications with require real-time behaviour.

The Erlang runtime environment allows programs compiled on any system to run on any other system with an Erlang runtime environment. The runtime system also allows code to be updated during execution without interrupting the program. This is usable when patching systems that cannot go down.

Unlike many other programming languages Erlang is a functional programming language. This approach is odd for many C programmers but at the same time easy to learn. The resulting code is easy to read and understand.

Erlang has many success stories. One of the latest is the AXD301 ATM switch. The software of the switch contains around 850000 lines of Erlang code and around 1000000 lines of C/C++ code\(^8\).

Erlang has been successfully ported to Solaris, BSD, Linux, Windows 95/98/NT/2000/XP and VxWorks. There are unofficial builds for MacOS X, IRIX and TRU64. Erlang is distributed in two versions; one open source version and one proprietary version.

\(^7\) Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams, 1996, Concurrent Programming in Erlang

\(^8\) www.erlang.org/faq/t1.html#AEN50
3.1.2 Erlang data types
Erlang supports a number of data types. These are:

- Constant data types:
  - Numbers: Example: 44, -21, 3.14159
    Numbers are divided into integers and floats.
  - Atoms: Example: my_atom, ‘My atom is a sentence’
    An atom begins with a lowercase letter.

- Compound data types:
  - Tuples: Example: {1, 2, mommy}, {{foo}, bar, 122}, {}
    Tuples are used to store a fixed number of elements in a structure.
  - Lists: Example: [], [1, 2, 3], [foo, 100]
    Lists are used to store a variable number of elements in a structure. Note that a list can contain different types of elements.

3.1.3 Concurrency and Distributed Systems
Erlang supports concurrency and IPC\(^9\) by default. There are no threads, only processes and the underlying implementation (weather it is threads or processes) is hidden. Processes can only communicate in one way; via Erlang message passing. Messages can be passed synchronously or asynchronously.

In a distributed Erlang system, processes are distributed within nodes. One process can send a message to a node or to a specific process within a node. Communicating with another node is as easy as communicating with a process within the same node. For security reasons a node which is to communicate with another node has to be aware of the so called cookie of the remote node. If it does not, it cannot communicate with the node. The cookie is represented as a string.

3.1.4 OTP
OTP (Open Telecom Platform) is a library for Erlang. It provides support for many common problems in network communication and telephony.

3.2 Software Architecture

This chapter describes some system design ideas. The ideas have been considered in the development of the test model.

3.2.1 General
Since systems tend to grow in size and complexity, organization of design get more and more important. This is where the principles of software architecture come into the picture. The issues of Software Architecture\(^10\) include:

- Composition of components.
- Communication protocols.
- Synchronization (Data access etc.).

\(^9\) Inter Process Communication
\(^10\) Mary Shaw, David Garlan, 1996, Software Architecture: Perspectives on an Emerging Dicipline
• Assignment of functionality to different subsystems.
• Composition of design elements.
• Physical distribution.
• Scaling and performance.
• Dimensions of evolution.

The described software architecture styles are taken from *Software Architecture* (Mary Shaw and David Garlan 1996).

### 3.2.2 Software Architecture Styles

There are several models for software architecture. Some of these styles will be briefly described here.

#### 3.2.2.1 Pipes and Filters

The Pipes and Filters style is well known to the experienced UNIX user. When working with shell programming, pipes are commonly used to make different programs or commands interact with each other. Consider the command:

```
$ finger foobar | grep Directory | head -n 1 | cut -d " " -f 2 | ls
```

This series of commands will list the contents of the home directory of the user foobar. The series of commands communicate via pipes and the commands can be considered filters.

One important property of the Pipes and Filters style is that the filters do not need to know about each other, therefore the filters become very independent. The only thing they have to agree is what is sent over the pipe.

Another good property of the style is the support for reusing filters. This property can be seen in the collection of UNIX commands that can be arbitrarily connected via pipes.

#### 3.2.2.2 Data Abstraction and Object-Orientation Organization

This is a style where the developer encapsulates data with its functions and hides the internal implementation of the data representation. Objects in a program can represent connections, database records, states etc.

An advantage of the style is that objects can change implementation without affecting the functionality of the system as long as the interface of the objects stays intact. A disadvantage is that objects have to be aware of other objects they communicate with.

#### 3.2.2.3 Event Based

Event Based systems are commonly used in GUI libraries like WinAPI and SDL. The idea of Event Based systems is that instead of invoking a method directly a message or event is broadcasted. The event can be handled by some other component of the system. This system has the advantage that it is often reusable. A component can be added to a new system by just letting other components adopt its events.

#### 3.2.2.4 Layered Systems

Layered Systems are systems organized hierarchically. Every layer is responsible for the interface to the above and the below layer.
One big advantage with this kind of style is that one layer can be replaced with another as long as the interface to above and below layers is kept constant. This style is used in the OSI ISO stack and the X-Window system. These systems show how another can replace a layer; for example, TCP can be replaced by UDP in the OSI model or KDE can be replaced by Fluxbox in the X-Window system.

### 3.2.2.5 Repositories
Repositories are built up of a central source of data holding some shared information and/or current state and clients accessing the data. The data source is called repository and can be a database, CVS repository etc.

### 3.2.2.6 Interpreters
This style is based on the idea that an interpreter interprets a defined formal language. The interpreter itself holds the current state of the executing program.

Java™ is based on the Interpreters idea. Java™ is built upon a virtual machine interpreting the assembler like code generated by the compiler.

### 3.2.2.7 Process Control
Process Control is a style where a controller continuously monitors the output of the process and the input to the process is regulated based on the output.

### 3.2.2.8 Distributed Processes
Processes distributed over a network share some common task divided into sub-tasks. This style is used in modern computer clusters. Tasks will be distributed to different processes using an algorithm like “Heartbeat” or “Bag-of-tasks”.

### 3.3 UML
This chapter describes UML in general and why it is used. UML is used later on to describe software and system design.

#### 3.3.1 General
UML 11 (Unified Modelling Language) is a standardized language for documenting software development. It contains a number of graphical elements, which are combined into diagrams. Since UML is a formal language it contains rules how these graphical elements are to be combined and connected.

UML describes what a system is supposed to do, not how it does this. Therefore the UML-model of a system does not tell how the system is implemented.

#### 3.3.2 Types of diagrams
UML contains many different types of diagrams. It is also make hybrids of these diagrams. The diagrams are:

- Class diagram: Components of the systems divided into classes
- Object diagrams: Instances of classes in a diagram.

---

11 Mark Priestley, 2003, Practical Object-Oriented Design with UML
• Use Case Diagram: A Use Case Diagram describes the system from the users point of view.
• State Diagram: A diagram describing the different states of the objects in the system. Contains a start state and an end state.
• Sequence Diagram: Describes how objects communicate over time.
• Activity Diagram: Describes the activity occurring within an object’s behaviour.
• Collaboration Diagram: Describes how the components of the system achieve the over all system objective.
• Component Diagram: Describes how the development teams cooperate and what their responsibilities are.
• Deployment Diagram: Describes the physical architecture of the system. Shows how different computers are connected and what software is running on each computer.

3.4 CORBA
This section describes CORBA\textsuperscript{12}; its capabilities, advantages and disadvantages. It is described here since it is referred to in other sections.

3.4.1 Concept
CORBA is a standard for accessing objects, distributed or local, implemented in different languages and running on different operating systems. Most major programming languages support CORBA, including Erlang.

One great advantage with CORBA is that it supports distributed objects and computing out-of-the-box. One of the basic ideas of CORBA is that it is equally as simple to call a method on a local as on a remote object. This can be used to access different functionality on different computers with full transparency.

CORBA interfaces are specified using the IDL language, which is a unified language for specifying how methods are called and what they return.

```cpp
interface salestax {
    float calculate_tax(in float taxable_amount );
}
```

The above example describes an interface named salestax with a method named calculate_tax, which takes one floating-point argument, taxable_amount, and returns a floating-point number.

Implementing this in C++ would result in the following code:

```cpp
class salestax : public CORBA::Object {
    virtual void calculate_tax(CORBA::Float taxable_amount) = 0;
};
```

The method is purely virtual since the specification declares an interface.

3.4.2 Hands on Example
This section explains implementation of a CORBA client and server written in C.

\textsuperscript{12} Common Object Request Broker Architecture
3.4.2.1 IDL-file

To implement CORBA support for a method an IDL interface is first created. This interface defines a module named Calc with an interface Calculator. The interface holds two methods, add and sub which are supposed to add and subtract two real numbers and return the result.

```idl
module Calc {
    interface Calculator {
        double add(in double number1, in double number2);
        double sub(in double number1, in double number2);
    };
}
```

To create a stub and C skeleton for the IDL-file an IDL-compiler is used. In this example the compiler used is orbit-idl, and the IDL file is named calculator.idl. To compile the file issue the following command:

```
orbit-idl -skeleton-impl calculator.idl
```

This will create all the files needed for the stub and the client as well as empty implementation of the methods defined in the IDL-file.

For details on how to implement the client and server in C see Appendix A.

3.5 Formal Languages and Syntax Analysis

This chapter describes some theory about formal languages and syntax analysis. Formal languages are typically programming languages, mathematical expressions etc.

Theory about formal languages is mentioned here since a formal language is defined in the final solution of the project.

3.5.1 Definition of a Formal Language

A formal language is a set of strings from an alphabet $\Sigma$, a finite set of symbols. We define $\Sigma^i$ as the all string containing $i$ symbols from $\Sigma$, $\varepsilon$ as an empty string and $\Sigma^*$ as:

$$\Sigma^* = \varepsilon \cup \Sigma^1 \cup \Sigma^2 \cup \ldots \cup \Sigma^\infty$$

A formal language is defined as:

$$L \subseteq \Sigma^*$$

A formal language can be defined by enumerating all strings in the language. This is however an ineffective way of defining a language. A more practical way of describing a language is to define rules for how strings in the language can be constructed or to describe an algorithm that determines whether a string is in the language or not.

3.5.2 The Chomsky Hierarchy

All languages can be classified in the Chomsky Hierarchy (Table 1). There are four different levels in the hierarchy. The level of a language defines how hard it is to recognize the language. Only regular and context-free languages will be further discussed in this section.

<table>
<thead>
<tr>
<th>Chomsky Language Class</th>
<th>Grammar</th>
<th>Recognizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Regular</td>
<td>Finite-State Automation</td>
</tr>
<tr>
<td>2</td>
<td>Context-Free</td>
<td>Push-Down Automation</td>
</tr>
<tr>
<td>1</td>
<td>Context-Sensitive</td>
<td>Linear-Bounded Automation</td>
</tr>
<tr>
<td>0</td>
<td>Unrestricted</td>
<td>Turing Machine</td>
</tr>
</tbody>
</table>
3.5.2.1 **Regular languages**

Regular languages can be recognized by a regular expression. An example of a regular language is all positive even numbers in binary format:

\[ \Sigma = \{0,1\} \]
\[ L = \{0,10,100,110,1000,1010,1100,\ldots\} \]

The language containing all positive even numbers can be defined by the regular expression:

\[ (0\lor 1)^*0 \]

Regular languages can be recognized by deterministic finite-state automation (DFA). Finite state automation is a state machine with a finite number of states and determinism is where only one state can be active at a time. The state machine representing the regular expression above is:

![Figure 4 State machine representing regular expression (0|1)*0](image)

In the state machine above (Figure 4), state `LastIsZero` is an accepting state and `InitialState` is the initial state.

3.5.2.2 **Context-free languages**

Some languages cannot be defined by regular expressions. One example of such language is the language of palindromes of strings containing 0 and 1. Assume:

\[ \Sigma = \{0,1\} \]
\[ L = \{0,100,11,000,101,010,111,\ldots\} \]

This language cannot be recognized by a finite state machine. This is because the finite state machine lacks memory. However, push-down automata (NPDA) can recognize a context-free language.
In the NPDA above all edges are marked with a condition. The meaning of this condition is:

\[
\text{[Read token, Stack top / Operation]}
\]

In the figure, $\#$ is used to mark empty stack and $e$ is used to mark $\epsilon$ (no token).

Since a regular expression cannot define a context-free language a more powerful definition is needed. We define a context-free language using a grammar:

\[
\begin{align*}
L & \rightarrow 0L0 \\
L & \rightarrow 1L1 \\
L & \rightarrow 0 \\
L & \rightarrow 1 \\
L & \rightarrow \epsilon
\end{align*}
\]

This grammar is said to be context-free since the left-hand side contains only one symbol.

### 3.5.3 Scanners

A scanner is the first stage used when processing a formal language. A scanner is a program unit which defines all the valid tokens in a language. A token represents any special character, word, number or string. It refers to a single symbol in the grammar.

The scanner scans the input for valid tokens and reports the non valid tokens. This stage is called **lexical analysis**. The output of this stage is a list of tokens.

### 3.5.4 Parsing

The parser is the next stage when processing a formal language. It recognizes the syntax of the language and builds an abstract syntax tree (AST) from the input (Figure 6).

![Figure 6 An AST from an expression.](image)
3.5.5 Type checking

The next stage in processing a language is type checking. In this stage a symbol table of all the variables, functions and types are built. This is done by traversing the AST bottom-up. For example, when parsing the AST above, the ‘a’ node is reached, this variable is checked against the symbol table and if it exists the stage is accepted. Then reaching the node ‘=’ the two children of the node are compared so that they are of the same type.
4 Solution

This section describes different approaches to a new test environment which satisfies the criteria; simplicity, reusability and flexibility.

4.1 System Design

This chapter describes a few ideas that can be used when designing a test system. The approach is to build an API which can glue together the test suite, the test instruments and test object. All equipment shall be controllable from the same test suite, no matter what the implementation language is.

4.1.1 Layered Design

One solution to the problem of simplifying testing is to build a hierarchical layered system. The layers in the model would be (Table 2):

<table>
<thead>
<tr>
<th>Layer</th>
<th>Layer task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test suite</td>
<td>Implementation of a test suite in some language.</td>
</tr>
<tr>
<td>API Wrapper</td>
<td>A Wrapper for the selected (test suite) language.</td>
</tr>
<tr>
<td>API</td>
<td>API that holds the actual implementation of the functionality in the model.</td>
</tr>
<tr>
<td>Communication</td>
<td>Communication layer to communicate with test objects and testing equipment.</td>
</tr>
<tr>
<td>Test objects /</td>
<td>The actual equipment being tested and the equipment being controlled from</td>
</tr>
<tr>
<td>equipment</td>
<td>the testbench.</td>
</tr>
</tbody>
</table>

Table 2 Table of the suggested layers.

This model applies to the model described by Mary Shaw and David Garlan in Software Architecture\textsuperscript{13}.

In the model each layer can hide itself from all other layers except for the adjacent outer layer. Sometimes some functionality may be exported from one layer to a layer not adjacent. The interfaces between the layers are defined by protocols. Since the interfaces are always intact one layer can be replaced by another without rewriting the adjacent ones.

Another property of the layered system design, which makes it a suitable design, is that it allows interaction on different levels of abstraction. For example, a layer can be placed between the test suite layer and the API Wrapper layer to implement functionality on higher level of abstraction than on the API Wrapper layer.

The major disadvantage of this design is the API Wrapper layer. This layer has to contain an implementation of the interface to the API for every programming language used for test suite development. In many cases this would consist of thin wrappers for different languages.

4.1.2 Pipes and Filters

Another design idea is to build a library of independent filters. The filters will work much like a network protocol, which packs the information into a packet and sends it to the next filter. Filters are unaware of each other and only responsible for filtering the payload and passing it on to the next filter.

This model applies to the model described by Mary Shaw and David Garlan in Software Architecture\textsuperscript{14}.

\textsuperscript{13} Mary Shaw, David Garlan, 1996, Software Architecture: Perspectives on an Emerging Discipline

\textsuperscript{14}
The major benefit of this design is that it supports reuse. If a test case must be built up by opening an ssh connection to a computer and from that open another ssh connection, the ssh filter will just be used twice. Putting a test suite together is a matter of connecting filters via pipes. How the pipes are implemented is an implementation detail and will not be adressed here.

One disadvantage of this idea is that the different filters will have to have different interfaces for different languages. The tester may want to connect his Erlang or C test suite to an arbitrary filter. Therefore each filter needs an interface for each language supported. This requires a lot of work for each new filter. One solution to this problem could be to write a filter to convert from any language to a specific communication protocol used between filters. This would make adding new filters more simple.

### 4.1.3 Conclusion

The most natural way of implementing the API is to use the layered approach. This is because each instrument differs from others in so many ways that the filters proposed in the Pipes and Filters approach would not be reused. The final design will consist of 5 layers (Table 3):

<table>
<thead>
<tr>
<th>Layer name</th>
<th>Layer task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test suite</td>
<td>The actual test suite implemented in C, Erlang, Tcl or Perl.</td>
</tr>
<tr>
<td>Wrapper Layer</td>
<td>A layer used to wrap the test suite language to the API implementation layer.</td>
</tr>
<tr>
<td>API</td>
<td>The API implementation layer implemented in any suitable language like C, Erlang, Tcl or Perl.</td>
</tr>
<tr>
<td>Communication layer</td>
<td>The communication to the instrument either via vendor provided API</td>
</tr>
<tr>
<td>Instrument API</td>
<td>or via developed communication methods.</td>
</tr>
<tr>
<td>Instrument</td>
<td>The actual instrument or test object to control.</td>
</tr>
</tbody>
</table>

Table 3 Table of the layers to be used.

Table 3 describes which layer to be used and the task of the different layers. Figure 7 show how the layers are connected and communicate.

This is the design to be used. However, one great problem remains; how is the Wrapper Layer to be implemented. This will be adressed in the next section.

14 Mary Shaw, David Garlan, 1996, Software Architecture: Perspectives on an Emerging Discipline
4.2 Communication in Wrapper Layer

This section describes different approaches to implement the communication over the Wrapper Layer.

4.2.1 CORBA

CORBA is one solution for the Wrapper Layer. Since CORBA supports all required languages out-of-the-box. No further language wrapping is needed.

If a CORBA method is to be called from shell scripts a manual mapping has to be done. This can be achieved by a 1-to-1 mapping between functions and programs.

4.2.1.1 CORBA pitfalls

CORBA fully supports all the required features of the project. It has full support for communication with all the required languages. It has also got some features not needed, but good for the project. Since it is network transparent the functionality can be implemented in the appropriate place.

However, selecting CORBA for interface implementation is not a trivial design decision. It will most certainly complicate the implementation of the API by adding extra complexity. The test suite side will be fairly complex in the CORBA case.
4.2.2 Sockets
CORBA is an implementation of remote procedure calls over sockets. Since it is a fairly com-
plex system a similar but simpler system can be suitable. One idea is to replace all the
CORBA interfaces with sockets and let the testbench communicate with the API by sending
messages over the TCP/IP socket. This implies that a protocol for transmitting messages from
the testbench to the API and vice versa has to be specified. Proposal of protocols will be de-
scribed later on in this section.

Almost all programming languages contain support for network programming.

4.2.2.1 Simple Line Oriented Protocol
The simplest way of protocol is to send messages as plain text over the socket. Each packet is
separated by a newline. When a method is to be called a message is sent over the socket con-
taining:

| function_name argument_1 argument_2 ... argument_n |

The API specifies how the arguments are to be sent (as integers, strings or decimal numbers)
and handles type mismatch errors.

For each method call (message from testbench to API) one reply is generated. The return
value consists of a tuple consisting of either:

| ok: Returnvalue |
| or:             |
| error: Reason   |

4.2.2.2 XML Protocol
Another more complex protocol type is to use XML. XML allows us to specify types and
compound types. The following language describes the types compound types and method
calls.

| string   | -> | <s>the string</s> |
| integer  | -> | <i>1</i>            |
| list     | -> | <l>2 3 4</l>        |
| tuples   | -> | <t>5 6</t>          |
| method call | -> | <m>               |
|         |   | <s>method_name</s> |
|         |   | <t>arguments</t>  |
|         |   | </m>              |
| method reply | -> | <r> |
|         |   | <i>status</i>     |
|         |   | <t>return value</t> |
|         |   | </r>              |

A message can be sent over the socket as plain text. To specify what is sent two fields are
added, the content-type and the content-length. The following example describes a call to the
method setup_hostname, taking a hostname (string) and an IP-address (tuple of integer) as
arguments.

| Content-type: text/xml |
| Content-length: 138   |
Both C and Erlang have implementations of XML parsers. This protocol is very similar to the XML-RPC, which will be discussed in the following section.

### 4.2.2.3 Socket Pitfalls

Sockets are a well known concept to all the test suite developers. However, the encapsulation is not all that good since it is hard to make type checking at compile-time.

### 4.2.3 XML-RPC

XML-RPC\(^\text{15}\) is a standard for remote procedure calls over TCP/IP. It uses HTTP as transport protocol and XML as encoding. An XML-RPC request is a HTTP POST request where the content is an XML page. This allows complex data structures to be transmitted. XML-RPC is supported by many programming languages such as Erlang, C, and Perl.

XML-RPC has many similarities with CORBA. In this case the advantages and disadvantages are practically the same as the advantages and disadvantages with CORBA.

### 4.2.4 Independent Programs

A different approach to the problem is to design a collection of independent programs for controlling equipment. These programs can share interfaces such as the meaning of different command line arguments and can be invoked from the test suite via shell commands. All major programming languages support invocation of shell commands; therefore programs will be accessible from all types of test suites (written in Erlang, C etc.) and command line. The collection of programs will also have compatibility with standard UNIX commands (such as GNU Core-utils).

The different programs can be written in any language, which makes the programs easy to implement but difficult to maintain.

One big advantage with this design is that the testers can use the programs both from test-benches and from shells. The usage of the programs is natural to the experienced UNIX user, and therefore the threshold of knowledge is not high when using a system like this. Many testers will probably consider the design natural and simple to use and develop.

One great disadvantage is that the system lacks the flexibility of the earlier described designs. Another disadvantage is performance. It is usually a slow operation to invoke shell commands from more low-level programming languages. However, in most cases the performance is not that important when writing test suites.

\(^{15}\) [www.xmlrpc.com](http://www.xmlrpc.com)
4.2.5 Erlang IPC communication

The major languages to support in the system are C and Erlang. One approach would be to base the implementation of the wrapper layer in these two languages. The standard Erlang distribution contains C-libraries for connecting to Erlang nodes, sending and receiving messages and converting primitive C-types to Erlang types and vice versa.

The methods that are to be executed in C are executed by a C-node and vice versa. To call a method in C a message is sent to the C-node to indicate what function to execute. The return values from the functions are sent back to the test suite by Erlang IPC.

4.2.6 Conclusion

The advantages and disadvantages of the different solutions are summarized in Table 4.

There are several systems similar to CORBA and XML-RPC. The reasons why these are selected in favour of other systems, like DCE etc, are that they are fully supported in Erlang\textsuperscript{16}. Erlang is a widely used programming language within Ericsson but practically unknown outside the organization. CORBA is also used within Ericsson to configure equipment such as switches, routers and gateways.

Both XML-RPC and CORBA are fairly complex solutions. The overhead work which has to be done to be able to call methods can be wrapped into more simple methods but the clients still tend to grow in complexity.

The socket solution is also fairly complex. However, the socket concept is known to all the testers and considered easy to understand.

The simplest solution is the Independent Programs approach. This approach is far less constrained and might result in many programs and almost no organization.

The best approach is to implement the wrapper layer based on Erlang IPC. This approach is fairly complex to implement but easy to use. One disadvantage is that it does not natively support other languages than C or Erlang. However since C-functions are easy to call from many other programming languages these can also be considered to be supported. Its biggest disadvantage is performance. On the other hand, performance is not considered an important issue in the current usage.

<table>
<thead>
<tr>
<th>Design idea</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORBA</td>
<td>• Well tested system.</td>
<td>• Complex</td>
</tr>
<tr>
<td></td>
<td>• Tester does not need to know anything about underlying implementation.</td>
<td></td>
</tr>
<tr>
<td>Sockets</td>
<td>• Easy and well known concept.</td>
<td>• No type checking.</td>
</tr>
<tr>
<td>XML-RPC</td>
<td>• Well tested system.</td>
<td>• Complex</td>
</tr>
<tr>
<td></td>
<td>• Tester does not need to know anything about underlying implementation.</td>
<td></td>
</tr>
<tr>
<td>Independent programs</td>
<td>• Simple to implement.</td>
<td>• Difficult to maintain.</td>
</tr>
<tr>
<td></td>
<td>• Easy to use.</td>
<td>• No type checking.</td>
</tr>
<tr>
<td>Erlang IPC</td>
<td>• Easy to use.</td>
<td>• Fairly difficult to implement.</td>
</tr>
<tr>
<td></td>
<td>• Easy to extend.</td>
<td>• Bad performance.</td>
</tr>
<tr>
<td></td>
<td>• Tester does not need to know anything about underlying implementation.</td>
<td></td>
</tr>
</tbody>
</table>

Table 4 Advantages and disadvantages of different wrapper layer implementations.

\textsuperscript{16} Anna Fedoriw, Bengt Nilsson, 2000, Erlang 5.0/OTP R7, CORBA and IDL Applications
4.3 Encapsulation of Wrapper Layer

Assume two methods are to be implemented; `IPnr2URL()` and `URL2IPnr()`. Assume that these methods are to be implemented in different languages (C and Erlang), and used from both languages. If there is a way to define which methods that are available, their parameters, return values and implementation language, all the function interfaces can be automatically generated.

When encapsulating the wrapper layer, Erlang IPC is used for the communication between Erlang and C. Of course another type of communication can be used.

4.3.1 Grammar

To define interfaces for methods a language is needed. This language must support different primitive types (integers and strings), compound types (tuples and lists) and type definitions. For each method its implementation language must be specified. For simplicity all methods and type definitions are placed in modules. The grammar will be:

```
<ModuleList>  : <Module> <ModuleList> |
| e
<Module>     : module <Identifier> {   
| <TypeDefinitionList> <FunctionList> 
} |
<TypeDefinitionList> : <TypeDefinition> <TypeDefinitionList> |
| e
<TypeDefinition> : typedef <Type> <Identifier> ; |
<FunctionList> : <Function> <FunctionList> |
| e
<Function>   : <Lang> : <QualifiedType> <Identifier> ( <Arguments> ) ; |
| <Lang> : Void <Identifier> (<Arguments> ) ; |
<Lang>       : C |
| Erlang
<Arguments>  : <QualifiedType> <Identifier> <ArgumentList> |
| Void
<ArgumentList> : , <QualifiedType> <Identifier> <ArgumentList> |
| e
<TupleList>  : , <QualifiedType> <Identifier> <TupleList> |
| e
<QualifiedType> : <Identifier> |
| Int |
| String
<Type>       : <QualifiedType> |
| <List> |
| <Tuple>
<Tuple>      : { <QualifiedType> <Identifier> <TupleList> } |
<List>       : [ <QualifiedType> ] |
<Identifier>  : IDENTIFIER
```

Since the grammar is context-free it can be parsed using a parser written in Flex and Bison\(^\text{17}\).

The interface file for describing the two methods `IPnr2URL()` and `URL2IPnr()` would be:

```haskell
module mod {
    typedef {
        Int m1,
        Int m2,
        Int m3,
        Int m4
    } IPnr;
    typedef String URL
    C:URL IPnr2URL(IPnr ipnr);
    Erlang:IPnr URL2IPnr(URL url);
}
```

To be able to parse the interface description and output function wrappers for Erlang and C, a parser is to be built.

### 4.3.2 Components

Each interface description will produce a number of components. Assume the following interface:

```haskell
module mod {
    typedef Int status;
    C: status startCEquipment(Void);
    Erlang: status startErlangEquipment(Void);
    Erlang: status startBothEquipment(Void);
}
```

The components of the output and the placement of these methods are described in Figure 8.
4.3.3 Parser output

The interface description will produce method interfaces for both Erlang and C. Depending on implementation language for the method it will do different things. Since the implementation language is either C or Erlang an implementation of a method will always be available. This implies that when the parser is producing the trivial implementation of the methods in C, all methods that are to be implemented in Erlang are implemented by sending a message to an Erlang node, that contains the non-trivial implementation, and returning the reply from the node. When the trivial implementation is produced for a method in C the implementation is just to call the method and return the return value.

Each module in an interface description produces a number of files. These files are described in Table 5.
### Table 5 Output files from parser.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>C files</td>
<td></td>
</tr>
<tr>
<td>[modulename].h</td>
<td>C-interface for all the methods defined in the module.</td>
</tr>
<tr>
<td>[modulename].c</td>
<td>Trivial C-implementation of the methods in the module.</td>
</tr>
<tr>
<td>[modulename]_types.h</td>
<td>Types defined in module.</td>
</tr>
<tr>
<td>[modulename]_typeconv.h</td>
<td>Header file for type conversion functions.</td>
</tr>
<tr>
<td>[modulename]_typeconv.c</td>
<td>Implementation file for type conversion functions.</td>
</tr>
<tr>
<td>[modulename]_impl.h</td>
<td>Header file for the implementation file.</td>
</tr>
<tr>
<td>[modulename]_impl.c</td>
<td>Non-trivial C-implementation of the functionality in the interface.</td>
</tr>
<tr>
<td>server.c</td>
<td>A server software emulating an Erlang node used to reply method requests. Only one C-server will be produced for all the modules.</td>
</tr>
<tr>
<td>Erlang files</td>
<td></td>
</tr>
<tr>
<td>[modulename].erl</td>
<td>Trivial Erlang implementation of the methods in the module.</td>
</tr>
<tr>
<td>[modulename]_impl.erl</td>
<td>Non-trivial Erlang-implementation of the functionality in the interface.</td>
</tr>
<tr>
<td>server.erl</td>
<td>An Erlang-node used to reply method requests. Only one Erlang-server will be produced for all the modules.</td>
</tr>
</tbody>
</table>

### 4.3.4 Types

The types defined in the grammar are defined with respect to Erlang types. All the types in the grammar will work natively in Erlang with one exception: in the interface description lists are only allowed to contain data of one type. In C however, the types have to be defined. The following section describes how the types in the grammar are translated into C.

\[
\begin{align*}
\text{Int} & \Rightarrow \text{int} \\
\text{String} & \Rightarrow \text{char *} \\
\text{typedef \{T1 m1, T2 m2, ..., Tn mn\} T;} & \Rightarrow \text{typedef struct T \{} \\
& \quad \text{T1 m1;} \\
& \quad \text{T2 m2;} \\
& \quad \quad \ldots \\
& \quad \text{Tn mn;} \\
& \quad \} T; \\
\text{typedef \[T\] Tlist;} & \Rightarrow \text{typedef struct Tlist \{} \\
& \quad \text{size_t m_length;} \\
& \quad \text{T *m_data;} \\
& \quad \} Tlist;
\end{align*}
\]

### 4.3.5 Node Communication

Erlang IPC will be used for communication in the demonstrative implementation of the solution. To call C functions from Erlang and vice versa, Erlang IPC messages have to be sent between the API and the server. Messages are defined as tuples containing information about the sender, which function in which module that is to be executed and arguments to that function.

If a C test suite is to call an Erlang function the C side sends a message to the running Erlang node. If the Erlang function needs to call a function on the C-side it sends a message to the C-node. This implies that the Erlang-node and C-node have to be running simultaneously. The server also has to be able to handle multiple connections since the number of function calls between the Erlang-node and C-node can be large.
If the interface described in 4.3.2 is implemented and used, the communication between nodes and API is described by Figure 9.

**4.3.5.1 Function call message**
A message for calling a function is defined as:

\[(\text{From}, \text{Module}, \text{Function} \{\text{Argument1}, \text{Argument2}, \ldots, \text{Argumentn}\})\]

**4.3.5.2 Function return message**
A function can return several different types of messages:

\[\{\text{ok}, \text{Returnvalue}\}\]

or:

\[\{\text{error}, \text{Reason}\}\]

**4.3.6 Node Implementation**
The implementation of the nodes is described by the following pseudo-code:

```plaintext
while(true) {
    mess = receive_message();
    fork();
    if (i_am_child()) {
        returnvalue = mess.module:mess.function(mess.arguments);
        send(mess.from, returnvalue);
        die();
    }
}
```
4.3.7 Parser Implementation

Since the grammar described is context-free it can be recognized by a program written in Bison and Flex. Bison is a general parser generator which can produce a parser in C from a context-free grammar. Flex is a lexical analyzer compatible with Bison. It is used to perform pattern-matching on text.

The scanner scans the input files for tokens which are processed by the parser. The parser parses the tokens (interface description) and produces a syntax-tree. The syntax tree is traversed and a symbol table is built. All the types used in the interface description are looked up in the symbol table. The syntax tree is then used to produce the output, by traversing it. Traversing a syntax tree is easily done by implementing a Visitor pattern\textsuperscript{18} (see Figure 10). The visitor pattern allows developers to add functionality without editing existing code. This allows output for new languages to be easily added.

![Visitor Diagram]

**Figure 10** The general visitor pattern.

4.3.8 Building a visitor in C

Since C and Erlang are the preferred languages at Ericsson the visitor has to be written in a non Object-Oriented language. In this case the compiler including visitors is built in C.

Building the visitor pattern in C is slightly more difficult than building it in C++. There are many approaches but the approach described here is the one used in the project. Assume that we have built a syntax tree for an assignment from a value to a variable. The types are integers and floats.

---

\textsuperscript{18} Erich Gamma, Richard Helm, Ralph Johnsson, John Vlissides, 1994, Design Patterns: Elements of Reusable Object-Oriented Software
4.3.8.1 Visitor

The visitor is the base class for all the different types of visitors. It contains basic functionality for traversing the AST. The visitor is built up by a struct holding pointer to a struct containing pointers to all the member functions.

Assume a grammar that allows assignment from variables of different types from constants (integers and floats).

```c
struct VisitorVTable {
    void (*visitAssign)(struct Visitor*, const struct Assign*);
    void (*visitType)(struct Visitor*, const struct Type*);
    void (*visitFloat)(struct Visitor*, const struct Float*);
    void (*visitInt)(struct Visitor*, const struct Int*);
};
struct Visitor {
    struct VisitorVTable *m_vtable;
};
```

When inheriting a visitor an instance of a visitor is placed first in the inherited structure.

```c
struct MyVisitor {
    struct Visitor m_base;
    /* More members goes here. */
};
```

Assume that MyVisitor would like to perform some action when visiting a struct Int. The class then sets the visitInt pointer to some other function performing the specific task.

4.3.8.2 Visitable

The Visitable is a base class for all visitable objects (in our case struct Int and struct Float and struct Assign).

```c
struct VisitableVTable {
    void (*accept)(struct Visitable *, struct Visitor *);
};
struct Visitable {
    struct VisitableVTable *m_vtable;
};
struct Type {
    struct Visitable m_base;
    union {
        struct Int *m_int;
        struct Float m_float;
    }
};
struct Assign {
    struct Type m_ltype;
    struct Type m_rtype;
};
struct Int {
    struct Visitable m_base;
};
struct Float {
    struct Visitable m_base;
};
```
Each object has an accept function of its own. In this function the correct visit function (for
the visitable) of the visitor is called.

When traversing the AST and a struct Type is accepted, it calls accept() for the type it is
holding. This makes the visitor visit the actual type (struct Int or struct Float) that the
struct Type object is holding.

### 4.3.9 Demonstration Example

To demonstrate the functionality of the developed model a basic test case for the IS\textsuperscript{19} framework has been specified. The specification of the test case is:

**Pre conditions:**
- IS is up and running.
- Router tester is connected to IS.

**Post conditions:**
- No alarms are signalled.
- No traffic is lost.

**Test:**
1. Make a connection over IS.
2. Start traffic on Router Tester.
3. Wait for 5 seconds.
4. Stop traffic on router tester.

First we identify what functionality is needed to write the test program. The functionality
needed is:
- Start and stop traffic on router tester.
- Open and close a connection on IS according to a configuration file.
- Wait for 5 seconds.

The API for controlling the Router Tester is easiest to implement in Tcl. The Tcl functions are
called from C. The API for controlling IS is easiest to implement in Erlang. To make all the
functionality for Router Tester and IS available from both C and Erlang the following inter-
face is created:

```plaintext
module rt {
    C: Void startTraffic(Void);
    C: Void stopTraffic(Void);
}
module rct {
    Erlang: Void add(Int bladeno, String fname);
    Erlang: Void modify(Int bladeno, String fname);
    Erlang: Void subtract(Int bladeno, String fname);
}
```

This interface description is parsed using:

```
parser -s < interface.txt
```

\textsuperscript{19} Integrated Site
The parser outputs a number of files. Most of these files do not have to be edited. In this case only two of the generated files have to be edited. These are *rt_impl.c* and *rct_impl.erl* and shall contain the implementation of the functions specified in the interface file, written in Erlang and C respectively. The implementations of the API functions used are in Appendix B:

Writing test programs to perform the test is a matter of calling the methods in the correct order.

### 4.3.9.1 C Test Suite

```c
#include <stdio.h>
#include "rt_types.h"
#include "rt.h"
#include "rct_types.h"
#include "rct.h"

int main() {
    char *ConfigFile = "config_add_mod_1";
    rct_add(1,ConfigFile);
    rct_modify(1,ConfigFile);
    rt_startTraffic();
    sleep(5);
    rt_stopTraffic();
    rct_subtract(1,ConfigFile);
    return 0;
}
```

### 4.3.9.2 Erlang Test Suite

```erlang
-module(testbench).
-export([tc_1/0]).

tc_1() ->
    ConfigFile=config_add_mod_1,
    rct:add(1,ConfigFile),
    rct:modify(1,ConfigFile),
    rt:startTraffic(),
    timer:sleep(5000),
    rt:stopTraffic(),
    rct:subtract(1,ConfigFile).
```
5 Conclusion

This project has turned out to be difficult both politically and technically. If a new test environment is to be developed it does not only have to fulfil the original requirements; flexibility, simplicity, and modularity, it also has to be politically correct. The testers are quite conservative and if they are to change the way they work the new way has to be significantly better than the old, but not too different from the old.

The result of this project is a model that fulfils the predefined requirements. The strongest argument for using the model is that the testers who want to write test programs in C can continue to do so and the testers who want to use Erlang can do so. The testers who use C do not have to write a single line of Erlang and vice versa. This makes the model politically correct which has developed into a requirement over the project time. Another strong argument for using the results of this project is the reuse issue. If one unit develops an API for communicating with some equipment and implements the API in C it can be directly used in Erlang with almost no work at all. This is a factor that will save lots of work, time and money in the organization.
6 Recommendations

The testers at Ericsson seem interested in the project and are discussing how to use the project for future testing projects. There are a few things that have to be looked into before the system is fully implemented and used:

- Is the interface description language flexible enough?
- Ability to access data types in different modules?
- Does the language need more primitive types?
  - Floating point numbers?
  - Atoms? (see 3.1.2 Erlang data types)

Currently there are a few weaknesses in the implementation. Therefore it is recommended to look over the current limitations before continuing the work. These limitations are:

- A function written in C, called by an Erlang test suite, cannot call another Erlang function.
- A function can only have one return value. However a function can return a tuple or a list containing several elements.
- All arguments are passed by value on the C side. Arguments shall instead be passed as pointers to constant objects. This saves time and memory.
- No error handling on the C side. Only return values from function can indicate errors; no message is available at the moment.

It is also recommended to examine another approach of communication in the wrapper layer. Currently only Erlang-IPC is implemented but the current parser implementation makes it easy to redefine the underlying communication mechanism. This does not mean that the other alternatives of communication are better but since time is limited in this project the alternative ways of communication have not been fully examined.

Currently the language for describing interfaces support C++ style comments. It would be possible to develop a general way of documenting the defined interfaces. This would allow the parser to automatically generate documentation in some form (HTML, PostScript or PDF) in a doxygen\textsuperscript{\textsuperscript{20}} manner.

\textsuperscript{20} http://www.doxygen.org
References

Books


ANNA FEDORIW, BENGT NILSSON, THE OPEN TELECOM PLATFORM PROJECT, 2000, Erlang 5.0/OTP R7 – CORBA and IDL Applications, Ericsson AB

DON LIBES, 1995, Exploring Expect, O’Reilly & Associates

ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES, 1994, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley


Ericsson Internal Documents


SIMON OLOFSSON, Blade MP - Logical Networks and IP Addressing, 2004, EAB/UPD/S.

STAFFAN BLAU, SBG Architecture & Concepts, 2005, EAB/UPS.


Internet

http://www.doxygen.org/, Last visited: 2005-11-08


Appendix A:
CORBA Client and Server

This appendix contains C-source for CORBA client and server.

C Server

The orbit-idl will create calculator-skelimpl.c, which is a file containing the implementation of the methods add and sub.

```c
static CORBA_double impl_Calc_Calculator_add(
    impl_POA_Calc_Calculator * servant,
    CORBA_double number1,
    CORBA_double number2,
    CORBA_Environment * ev) {
    CORBA_double retval;
    retval = number1 + number2;
    return retval;
}

static CORBA_double impl_Calc_Calculator_sub(
    impl_POA_Calc_Calculator * servant,
    CORBA_double number1,
    CORBA_double number2,
    CORBA_Environment * ev) {
    CORBA_double retval;
    retval = number1 - number2;
    return retval;
}
```

The server also needs a main method to run.

```c
#include <stdio.h>
#include <assert.h>
#include "calculator-skelimpl.c"
#include "exception.h"

int main(int argc, char* argv[]) {
    CORBA_ORB orb;
    CORBA_Environment* ev;
    Calc_Calculator calculator;
    PortableServer_POA root_poa;
    PortableServer_POAManager pm;
    CORBA_char* objref = NULL;
    ev = g_new0(CORBA_Environment, 1);
    CORBA_exception_init(ev);
    /* Init ORB, argc shall contain IOR string. */
    orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", ev);
    Exception(ev);
    root_poa = (PortableServer_POA)
        CORBA_ORB_resolve_initial_references(orb, "RootPOA", ev);
    Exception(ev);

    /* ...
```
C Client

The client also needs a main function to operate.

```c
#include <glib.h>
#include <orb/orbit.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "calculator.h"
#include "calculator-stubs.c"
#include "exception.h"

Int main(int argc, char* argv[]) {
    /* CORBA environment variable. */
    CORBA_Environment *ev;
    CORBA_ORB orb;
    CORBA_Object server;
    CORBA_double res;
    int dummy_argc = 1;
    char* dummy_argv[] = {argv[0], 0};
    if (argc != 2) {
        printf("Usage: %s IOR-string\n", argv[0]);
        return 0;
    }
    if (strncmp(argv[1], "IOR:", 4) != 0) {
        printf("Argument: \n%s\ndoes not look like an IOR. It does not
begin with \"IOR:\", argv[1]);
        return 0;
    }
    /* Allocate a new environment. */
    ev = g_new0(CORBA_Environment, 1);
    CORBA_exception_init(ev);
    /* Init the ORB. argc will contain an IOR string. */
    orb = CORBA_ORB_init(&dummy_argc, dummy_argv, "orbit-local-orb", ev);
    CORBA_ORB_run(orb, ev);
    return 0;
}
```
/* Translate the IOR string to an ORB object. */
server = CORBA_ORB_string_to_object(orb, argv[1], ev);
Exception(ev);

/* Call one method on the server. */
res = Calc_Calculator_add(server, 1.0, 2.0, ev);
Exception(ev);
printf("1.0 + 2.0 = %f\n", (float)res);
res = Calc_Calculator_sub(server, 1.0, 2.0, ev);
Exception(ev);
printf("1.0 - 2.0 = %f\n", (float)res);

/* Release the handle. */
CORBA_Object_release(server, ev);
Exception(ev);
return 0;
Appendix B: Router Tester and IS API Implementations

rt_impl.c

#include <tcl.h>
#include "rt.h"
Tcl_Interp* getHandle(void) {
    static Tcl_Interp *handle = NULL;
    if (handle == NULL) {
        handle = Tcl_CreateInterp();
        if (Tcl_Eval(handle, "source RT-lib.tcl")) {
            printf("Some error occurred loading RT-connect.tcl. message is: %s\n", handle->result);
        }
    }
    return handle;
}

enum status impl_rt_startTraffic() {
    return (TCL_OK == Tcl_Eval(getHandle(), "RT_startTraffic")) ? ok : error;
}

enum status impl_rt_stopTraffic() {
    return (TCL_OK == Tcl_Eval(getHandle(), "RT_stopTraffic")) ? ok : error;
}

RT-lib.tcl

source ./AgtClient.tcl
package require AgtClient 1.0
proc RT_getSessionID { } {
    set USER [exec printenv USER]
    set N2XHOST [exec printenv N2XHOST]
    set agtSetServerHostname [AgtSetServerHostname $N2XHOST]
    set agtListOpenSessions [AgtListOpenSessions]
    set found 0
    foreach agtOpenSession $agtListOpenSessions {
        set agtGetSessionLabel [AgtGetSessionLabel $agtOpenSession]
        if {$agtGetSessionLabel == $USER} {
            set found 1
            break
        }
    }
    if $found==0 { exit -1 }
return $agtOpenSession
}
proc RT_startTraffic ( ) {
    AgtInvoke AgtTestController StartTest
}
proc RT_stopTraffic ( ) {
    AgtInvoke AgtTestController StopTest
}
proc RT_getStreamNames ( ) {
    return [AgtInvoke AgtStreamGroupList ListNames]
}
# Constructor
proc RT_init ( ) {
    AgtConnect [RT_getSessionID]
}
proc RT_destroy ( ) {
    AgtDisconnect
}
# Make sure we run the constructor.
RT_init

rct_impl.erl

-module(rct_impl).
-compile(export_all).

get_config(Config,File) ->
    {ok,ConfigList} = file:consult(File),
    case lists:keysearch(Config, 1, ConfigList) of
        {value, {_,Type}} -> Type;
        false ->
        io:format(" Config  = ~p not found ~n",[Config]),
        false
    end.
add(BS,File) ->
call(BS,ch,brchDebug,forced_release_resources,[ip]),
    CtxtId=get_config(ctxtId1,File),
    Create_T1=get_config(create_T1,File),
    Create_T2=get_config(create_T2,File),
    Topology=get_config(topology,File),
    {call(BS, om, rctI, add, [CtxtId, Create_T1, Create_T2, Topology])}.
call(BsNo,Roles,M,F,A) ->
erlang:set_cookie(node(), get_cookie(BsNo)),
rpc:call(get_node_new(BsNo,Roles),M,F,A).
modify(BS,File) ->
    CtxtId = get_config(ctxtId1,File),
    Modify_T1 = get_config(modify_T1,File),
    Modify_T2 = get_config(modify_T2,File),
    {call(BS, om, rctI, modify, [CtxtId, Modify_T1, Modify_T2])}.
subtract(BS,File) ->
    CtxtId = get_config(ctxtId1,File),
    Subtract_T1 = get_config(subtract_T1,File),
    Subtract_T2 = get_config(subtract_T2,File),
%% Delete T1 and T2
{Ret, _} = call(BS, om, rctI, subtract, [CtxId, Subtract_T1, Subtract_T2]),
{Ret}.

get_cookie(BsNo) ->
    {ok, [ConfigList]} = file:consult(bladeConfig),
    case lists:keysearch(BsNo, 1, ConfigList) of
        {value, {BsNo, {Cookie, _, _}}} -> Cookie;
        false ->
            io:format("Config = ~p not found ~n", [BsNo]),
            false
    end.

get_node_new(No, Roles) ->
    {ok, [ConfigList]} = file:consult(bladeConfig),
    case lists:keysearch(No, 1, ConfigList) of
        {value, {No, {Cookie, Node1, Node2}}} ->
            case get_role(Node1, Roles, Cookie) of
                false -> get_role(Node2, Roles, Cookie);
                N -> N
            end;
        false ->
            io:format("Config = ~p not found ~n", [No]),
            false
    end.

get_role(Node, Role, Cookie) ->
    erlang:set_cookie(node(), Cookie),
    case rpc:call(Node, rcmI, cp_info, []) of
        {badrpc, nodedown} -> false;
        List -> find(Role, List)
    end.

find(_, []) -> false;
find(Role, [(PPB, [{Node}, _, _, {roles, RolesList}, _, _]) | T]) ->
    case list_find(Role, RolesList) of
        false ->
            find(Role, T);
        true -> Node
    end.

list_find(_, []) -> false;
list_find(Role, [Role|_Tail]) -> true;
list_find(Role, [_|T]) -> list_find(Role, T).
Appendix C:
Interface Parser Manual

This appendix contains a manual for the interface parser.

Manual for Parser

parser - Parse interface description files and output function headers for C and Erlang.

Synopsis

parser [OPTIONS] < interface.txt

Description

parser is a program developed to parse interface description files and produce function skeletons in C and Erlang. The output of the program is a number of files which will make all the functions callable from C and Erlang no matter what implementation language the function uses. Note that all previously created files will be overwritten.

Options

-v Run command in verbose mode.
-p Pretty print the input file to stdout.
-n Do not generate output. Only perform syntax and type check.
-s Generate implementation skeletons.
-h Print help message and exit.

Interface Language

The interface language supports a number of types. These are:

- Int (integer)
- String
- Tuple
- List

All types (except String and Int) have to be defined using typedef before use.

Example:

The syntax of the interface description is shown by this example.

```
module module_name {
    typedef {
        Int m1,
        Int m2,
        Int m3,
        Int m4
    } IPnr;
    typedef (IPnr ip1, IPnr ip2) IPnrPair;
    typedef Int bool;
```
As seen in the example above implementation language is defined per function. All types used as argument to functions have to be defined before use.

**Name Conventions**

The following rules define how modules, functions, types and arguments are named:

- A name matches the regular expression: `[a-zA-Z\-_]+[a-zA-Z0-9\-_]*`
- Each module has to have a unique name.
- Each data type has to have a unique data type name within its module.
- Each function has to have a unique function name within its module.
- Each argument to a function has to have a unique name.

This implies that the following interface is valid:

```c
module foo {
    typedef Int foo;
    C: foo foo(foo foo);
}
```

**Limitations**

- If you are using GCC as compiler no initialization for the library is needed, since it is done automatically. If another compiler is used to build the library, it has to be initialized by calling:
  ```c
  init();
  ```
- A function defined in an interface description and implemented in C cannot call a function implemented in Erlang. This limitation is known and will be fixed for future releases.

**Author**

Tomas Mannerstedt, contact: mannerstedt[at]gmail[dot]com