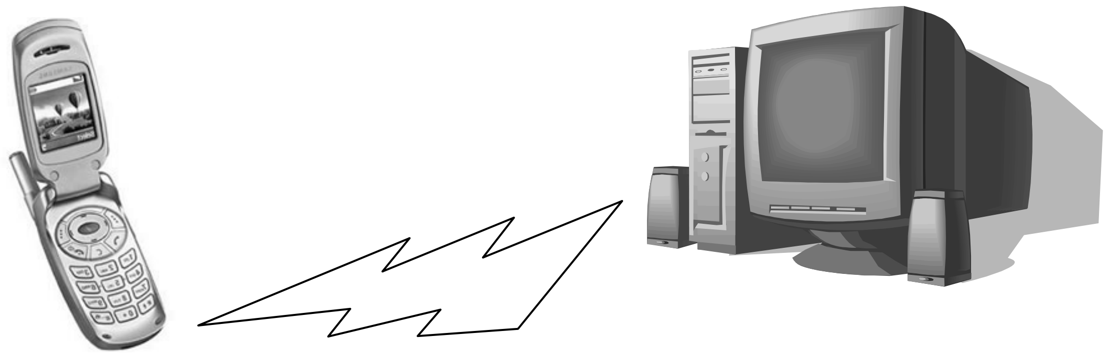
Automated Generation of Software Test Cases for Mobile Business Based on Tracing Mechanisms



Diploma Thesis by

Ronny Kießling

Author:	Ronny Kießling – 129086, University of Potsdam
Responsible University Teacher:	Prof. Dr. Helmut Jürgensen, University of Potsdam
Mentor:	Dr. Henning Schmidt, Texas Instruments Berlin AG
Processing Period:	2003/10/23 – 2004/02/23

Abstract

Topic:	Automated Generation of Software Test Cases for Mobile Business Based on Tracing Mechanisms
Author:	Ronny Kießling
Course of studies:	general computer science
Key words:	mobile business; test case generation; tracing; C++; XML

In this document a test case generator is presented which, unlike commonly known generator tools, does not use software specifications or implementations as input but rather data recorded during prior test runs. At first sight, this approach may sound a bit strange but various fields of application have been identified, e.g., for reproduction of erroneous behavior occurring only in very specific environments.

The actual implementation of a first version was done during the author's work at the company TEXAS INSTRUMENTS BERLIN AG. The software framework used there, as well as the logging tool (also developed by the author) is discussed in the context of theoretical conception and practical realization of the test case generator. Requirements concerning user interface, performance and reliability are analyzed and problems during the implementation are listed. Furthermore, possibilities to test the test tools themselves are examined.

Since the concrete environment of the proposed generator is the mobile business, a general introduction to software and testing methods used for cellular networks is given in the document, too.

Finally problems not yet solved are discussed and the planning for further development is presented.

Preface

Since the beginning of my study at the UNIVERSITY OF POTSDAM in 1995 I had in mind to finish it some day with a work that would not end up “covered in dust” but be of practical use in a certain field of the computer science. During almost four years of work as a student coworker at the company TEXAS INSTRUMENTS BERLIN AG I could experience the increasing importance of software test automation and I am full of hope that my beta-version of a test case generator will evolve to an essential tool for assuring the quality users expect from today’s mobile applications.

I’d like to thank Prof. Dr. H. Jürgensen and Dr. H. Schmidt for the highly skilled support but also for the patience they offered to me; my competent team-leader in the company, F. Reglin, for ideas which originally lead to my concept of a generator; A. Schmalwasser, among other colleagues, for giving important feedback as the first beta-tester; R. Radzinski, the English teacher who had to fight my “extravagant” grammar; the scientific worker O. Boldt, my brother André and many others not mentioned here for inspiring discussions; my fellow student Thomas for spending necessary relaxation breaks with me; Rene, my firefighter comrade, for providing his color laser printer; and last but not least my girlfriend Ina for accepting many lonely evenings and, nevertheless, advising me concerning the layout.

Ronny Kießling, February 2004

Table of Contents

List of Figures and Tables

1 Introduction

Many books have been written about *software*[1] *testing* since G. J. Meyers introduced it in [MYERS], 1979, as the part of software development taking up 50% of time and production costs but being less investigated then any other aspect. Even now, the ultimate testing strategy has not been found and due to obvious limitations alternatives are increasingly examined. Nevertheless *test* execution is still the method mainly used to find errors and ensure product quality. Unfortunately many software companies tend to release new product versions insufficiently tested, and annoyed end-users have to report the remaining errors. Such erroneous applications can lead to great monetary losses, thinking, e.g., of the introduction of the new tollage system on German autobahns in 2003, or even more terrible catastrophes.

To make the testing process more efficient automation becomes more and more important today. Although, as is stated in [POSTON], *"newcomers to automated software testing often think that all the automation they need is built into a capture-replay tool that runs or executes test cases"*, there are already a lot of commercial *tools* on the market which support automating the other technical activities – *test case* creation and evaluation. Generation of test cases is the topic of this paper but in contrast to common methods using, e.g., specifications the input for the proposed generator are data recorded during prior tests. This approach shall be understood as an addition to others rather then a replacement. In fact, it offers new possibilities which will be described later in the document.

This diploma thesis was written during my work as a software engineer at TEXAS INSTRUMENTS BERLIN AG[2] (*TI*, formerly CONDAT AG, see [TI] and

[1] Terms and abbreviations emphasized like this at their first appearance are described at the end of the document.

[2] Firm or product names capitalized like this are trademarks of the respective company.

[CONDAT]). As one main product, this company fabricates software for cellular phones. In the mobile business specific problems appear, like unpredictable network conditions, and timing takes an important role during operation. From Chapter 3 I concentrate on the highly specialized testing procedures which have been developed at TEXAS INSTRUMENTS.

Beside a theoretical introduction in software test automation, the main outcome of this work is a generator which produces test cases in a format that can directly be interpreted by existing test tools which are used regularly by developers and testers of the company. This generator examines log files created by another application (*PCO*) which has been developed by the author during a former undergraduate thesis (see [TRACING]).

Concretely the following topics will be presented:

– General overview about today’s software testing standards, methods and automation possibilities (Chapters 2)

– Study of the specifics in the mobile sector, introducing the *tracing* mechanism and the *TDC* language (Chapter 3)

– Explanation of the theoretical concepts underlying the test case generator developed by the author and the framework used (Chapter 4)

– Presentation of details concerning the implementation of the generator, its usage and comparing measurements (Chapter 5)

In Chapter 6, conclusions on the discussed topics will be drawn.

2 Software Testing – State of the Art

In this chapter the terms “software error” and “software testing” will be introduced. Furthermore, the various testing methods used today will be presented. Subchapter 2.4 gives an overview of specification types and *script* languages currently in use, followed by a subchapter which summarizes facts on how test processes can be automated. In the final section it will be clarified what makes up a “generator” and what kinds are available on the present market.

2.1 Software Errors

More and more parts of our daily life rely on computers – this means, in general, on *hardware* and software. To gain all the benefits the functioning of computer systems is expected to be error-free – which is too often not the case as anyone ever been in contact with a personal computer (*PC*) will confirm.
But while a crash of an office application during the writing of, e.g., a diploma thesis is most annoying the collapse of a space shuttle navigation system can result in a disaster:

On June 4th, 1996 in Kourou / Fr. Guyana, the first flight of the European space shuttle “Ariane 5” (weight: 740 t, payload: 7-18 t) was terminated 39 seconds after ignition – by self-destruction. The main problem had been an unexpected and unhandled value overflow. The navigation computer (running an *ADA* program) was considered to be reliable because it had already been used with “Ariane 4”. Development costs of 5 900 000 000 € (in 10 years) where “blown” into space. Fortunately the freight contained no humans – but 4 Cluster-Satellites. See [ESA] and [ARIANE] for more details; and [DISASTERS], [DISASTERS2], [DISASTERS3] for collections of other hazardous software bugs.

Why can software programs behave that erroneously? The following example gives an impression of how a small error can cause big problems. It is written in the widely used programming language “C”:

```
while (x > 0,1) { /* … */ }
```

This statement will lead to an endless iteration – in other words a controlled system would, e.g., stop or rotate infinitely. Apparently the author intended to compare x with the value “0.1”, but the typing mistake causes the consecutive evaluation of “x>0” and “1” where the latter will always be “true”. No human programmer is immune of such oversights. Of course this is only one possible source for software errors. See Chapter 4 of [KANER] for a detailed classification. A quite general definition for errors is the following:

”An error occurs if a system does not fulfill the requirements specified.“ (in [HORN])

Concerning software programs one basic requirement is stability, others depend on the field of application.

How is it possible to detect mistakes before they can do serious damage? To answer this question, software should first of all be delimited from hardware in terms of malfunctioning. A well known but exaggerating comment states:

"Hardware may fail, software is broken from the beginning!" (source unknown)

While hardware supports essentially the same small set of basic instructions stored permanently in each individual computer of a given type, software is not bound to a dedicated computer. Therefore a piece of hardware once proved to function correctly will do its job – at least for a longer time period. For software, several advantages seem obvious: Software cannot be worn off like physical components. This, of course, does not mean that it will function correctly after modifications of the environment, e.g., the operating system. Since textually stored software can be easily changed or adapted to upcoming needs. For more detailed examinations see [SOFTHARD], [SOFTWARE] and [HARDWARE].

With a continually increasing supply of instructions, libraries and components even very complex logics can be implemented in software – but with the draw-back of increasing source code to maintain. The following table (Table 1) gives a general overview of the dimensions already reached (see [BALZERT]):

TeX 82	14 000 lines of code
Cellular Phone	200 000 lines of code
Hubble Ground Software	1 000 000 lines of code
Atmosphere Control	2 000 000 lines of code
Space Shuttle, IIS	3 000 000 lines of code
B-2 Stealth Bomber	4 000 000 lines of code
Windows 95	10 000 000 lines of code
Windows NT 4.0	16 000 000 lines of code
Windows 2000	27 000 000 lines of code

Table 1 – Source Code Dimensions

One problem clearly appears: The more lines of code we have the higher is the probability that mistakes where made while writing them. This dilemma is, furthermore, compounded by the time pressure weighing on the developers due to the competition on the software market.

There exist many statistics concerning average error rates in today's software; see, e.g., in [NASA]. The following table (Table 2) gives some examples:

Standard Software	25 errors per 1000 lines of code
Important Software	2 - 3 errors per 1000 lines of code
Medical Software	0.2 errors per 1000 lines of code
Space Shuttle Software	< 0.1 errors per 1000 lines of code

Table 2 – Average Error Rates

To get a more intuitive impression [BALZERT] explains what an error rate of only 0.1% means for the western economy: 20 000 unusable medicaments or 300 defective pacemakers per year; 500 errors during medical operations per week; 16 000 lost pieces of mail or 18 airplane crashes per day; and 22 000 cheques booked incorrectly per hour! Apparently, life as usual would be impossible and something has to be done to keep the error rate as low as possible.

2.2 Software Testing

Still 50% of failures in industrial applications are caused by software errors. Nevertheless, the error rates have decreased significantly since the importance of computers for serious tasks started to grow. For example in 1977, on the average, 1 000 lines of code contained 7.0–20.0 defects; in 1994 only 0.2–0.05 where erroneous (see [BALZERT]).

How could this have been achieved? First, techniques for analyzing the requirements and creating the software design (e.g., prototyping) have been improved and, therefore, many errors could already be avoided during this phase. But, as a statistic in [TRAUBOTH] states, 36% of the software errors are introduced during the implementation phase, so it is worth looking at methods used in this stage as well. The concept of software testing was distinguished from pure *debugging* by 1957 and in the 1970s “software engineering” as a term was used more often (see [KIT]). In 1979 G.J. Meyers made his famous statement:

„Testing is the process of running a program with the intention to find errors.“ ([MYERS])

In other words it is not intended to prove the absence of errors which is in fact not possible by executing tests because of the undecidability of the “halting problem” (see [SIPSER]). G.J. Myers, furthermore, presents in [MYERS] a simple program, containing a loop and a few “if”-statements, which has 100 trillion execution paths. A fast tester could test them all in a billion years.

Currently several algorithms, like “Evolving Algebras” (see [EVALG]), exist with which formal correctness for small software modules can be shown. See [FORMAL] for an intuitive introduction to formal specifications. But even if all the complex details of a piece of software could be formally specified, correct behavior in a real environment (operating system, hardware, etc.) still cannot be guaranteed – unless all possible environmental conditions are formalized as well.

By software tests we try to prove that the part under test (e.g., a function or a module) is not performing the tasks as specified. So testing, in contrast to programming (implementing), is not a constructive activity but destructive instead – seen short-termed. To achieve positive results in the long run the work of testers should not be underestimated concerning time, effort and costs.

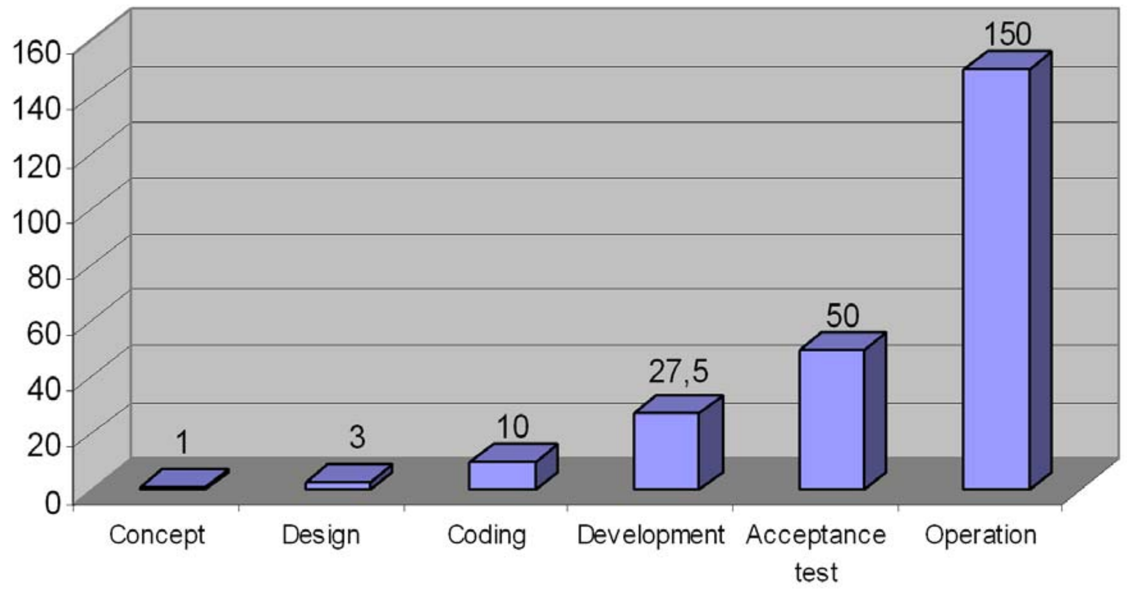


Figure 1 – Relative Costs in Different Development Phases

In general, the later a mistake is detected the higher is the expense of removing it. Figure 1 taken from [SCHIRM], presents empirical results concerning this problem. Avoiding mistakes while writing the source code would be best, of course. There are several development tools supporting the developer in various ways, e.g., by automatically extending reserved words, as done, for example, in the MICROSOFT DEVELOPER STUDIO.

Often incorrect functioning is caused by the structural characteristics of the program source. Thus further tests applied to the running program are indispensable. These should not only be performed by the developer itself to avoid biased results. At least, as is pointed out in [BEIZER], *"programmers must wear two hats: a programmer's hat and a tester's hat. When they are testing, they should [...] think like testers"*. There are, in fact, companies offering testing to software firms. Moreover, organizations like the INTERNATIONAL INSTITUTE FOR SOFTWARE TESTING (see [IIST]) or the GERMAN TESTING BOARD (see [GTB]) try to *"promote a disciplined approach to software testing and to caution against ad hoc testing by non-qualified individuals and groups."*

Although software tests can never guarantee 100% error-free functioning (see, e.g., [MYERS]) they help to approach that goal. Today's standard methods are described in the next subchapter.

2.3 Testing Methods

This section contains a compact overview of testing methods. See, e.g., [MYERS] and [KANER] for more detailed introductions. The key software testing standards can be found in the appendix of [KIT].

In general manual and computer-aided as well as static and dynamic methods are distinguished. To determine whether a system has behaved correctly on test execution the outcome of a test is usually predicted by a so-called *oracle* (see [HOWDEN]). Finding suitable oracles is a critical part of software testing.

Manual Tests: Manually the developer or a dedicated tester could check the source code for, e.g., unintended statements, like assignments where actually a comparison was meant. This static procedure is strongly supported by today's *compilers* which can detect many mistakes of that kind and will generate warnings and errors – depending on the specified error level.

The *Code Walkthrough* is an example for a dynamic, but manual method. Here the tester tries to imagine the paths the processor could take through the source code, seeking for problematic or erroneous situations. This can be combined with the developer explaining his[3] sources to other experts.

Many bugs (like missing or wrong variable declarations) can already be found using these techniques. But in most cases they will not substitute dynamic computer-aided methods applied to the actually running program or parts of it. For one thing with complex source code it is hard to find all possible program states beforehand. Timing conditions are another problem upcoming only while truly running the code. On the other hand, examples like the one described in [THERAC] demonstrate that exclusive usage of computer-aided tests is also not sufficient.

Computer-Aided Tests: Before performing such tests two main decisions have to be made: What shall be tested and how should it be done. Before the delivery of a product an acceptance test has to be performed, most likely in cooperation with the customer to demonstrate the principal functionality is as expected. This cannot cover all critical situations and has to be preceded by various independent tests of the individual components (modules and functions) but also by so-called integration tests where the cooperation of the components is checked. If all available components are involved we speak of system tests.

When testing the interoperability of several components, especially in the development phase but also to avoid too many possible error sources, *pseudo modules* can be used to emulate the environment.

[3] The masculine form will be used for both genders in this document without discriminating intensions.

Depending on whether we start testing small components or complete systems, either a *bottom-up* or a *top-down* strategy is used. It furthermore makes sense to rerun a dedicated test on the same software part after each modification of the program to ensure that behavior has not changed. Such techniques are called *regression tests* and should be automated as much as possible.

With all these tests, finally the proposed quality concerning performance, security and usability shall be ensured. Especially for end user applications, like office software, *alpha*- and *beta-test* phases (see [BETA]) are defined where, during the first one, the developer installs the application and tries to use it. The beta-test is performed by selected persons who should not have been involved in the development.

After having now clarified "what" has to be tested the possibilities of "how" a (part of a) running application can be tested will now be examined. We differentiate between *black-box* and *white-box* test conditions. A mixture of them called *gray-box* tests can also be found. In the first case the module or system under test is considered to be a sealed box with a clearly defined interface for accessing its functionality. The test consists of passing carefully chosen values as parameters to that box and comparing the returned values or triggered behavior with the specification. Such parameters can be found, e.g., by defining equivalence classes concerning valid and invalid values. Checking only one member of each class and combinations of them minimizes the number of tests but might leave some special problems, e.g., appearing only with a particular number which induces a division by zero, undiscovered. Another approach is to check extreme values regarding the *domain* limits. If, for example, the number of a month is expected, 0, 1, 12 and 13 are tried representing values just valid/invalid (see [HORN]). This method is also known as *data-driven* testing. See [BEIZER] for a comprehensive analysis of the black-box approach.

In contrast to the method described before during white-box testing the internals of the component under test are known to the tester. By taking the component's control flow graph as a base, tests can be arranged to check the statement, branch, condition and path coverage. *Function-driven* testing is another term describing this method. The debugging process, where the program is executed step by step to find errors, could also be assigned to the white-box rubric, but most scientist do not accept it as a testing technique.

To support the testing process program *instrumentation* can be used, e.g., generation of extra output. This often speeds up finding bugs but also influences the timing conditions. It is, therefore, quite common to keep instrumentations in the release to avoid retesting all components without them.

2.4 Specification and Test Script Languages

Software specifications state or picture how software is expected to behave. Additionally, operational characteristics like performance can be described (see [POSTON]). Not long ago a common opinion among software engineers was that it would be wasting time to record a description of how software was supposed to behave instead of directly coding it, as analyzed in [JONES], 1991. In the meantime software design became an increasingly important part of the development process and, therefore, it was necessary to record information concerning requirements systematically. Such specifications not only enable changes, e.g., because of new requests from customers, on a higher level; they also give testers a suitable reference to detect deviations from expectations. The so-called *One-Source-Concept* is a base strategy supported by many companies in these days – one source, the specification, for designers, programmers and testers. Unfortunately, this concept is often not rigorously applied.

One of the main reasons is that the specification is not complete or ambiguous. Various attempts were made to define standardized languages with which also semantic rules could be expressed. Such formal specifications are, moreover, much easier to handle by automation tools. Textual notations are usually designed in a way which makes them familiar to anyone who reads English. The *Semantic Transfer Language* (*STL*, in *IEEE* Standard 1175-1994, see [IEEE]) is one example. However, graphical specifications became increasingly popular. They are easier to understand by most people and software tools exist, able to directly take them as input.

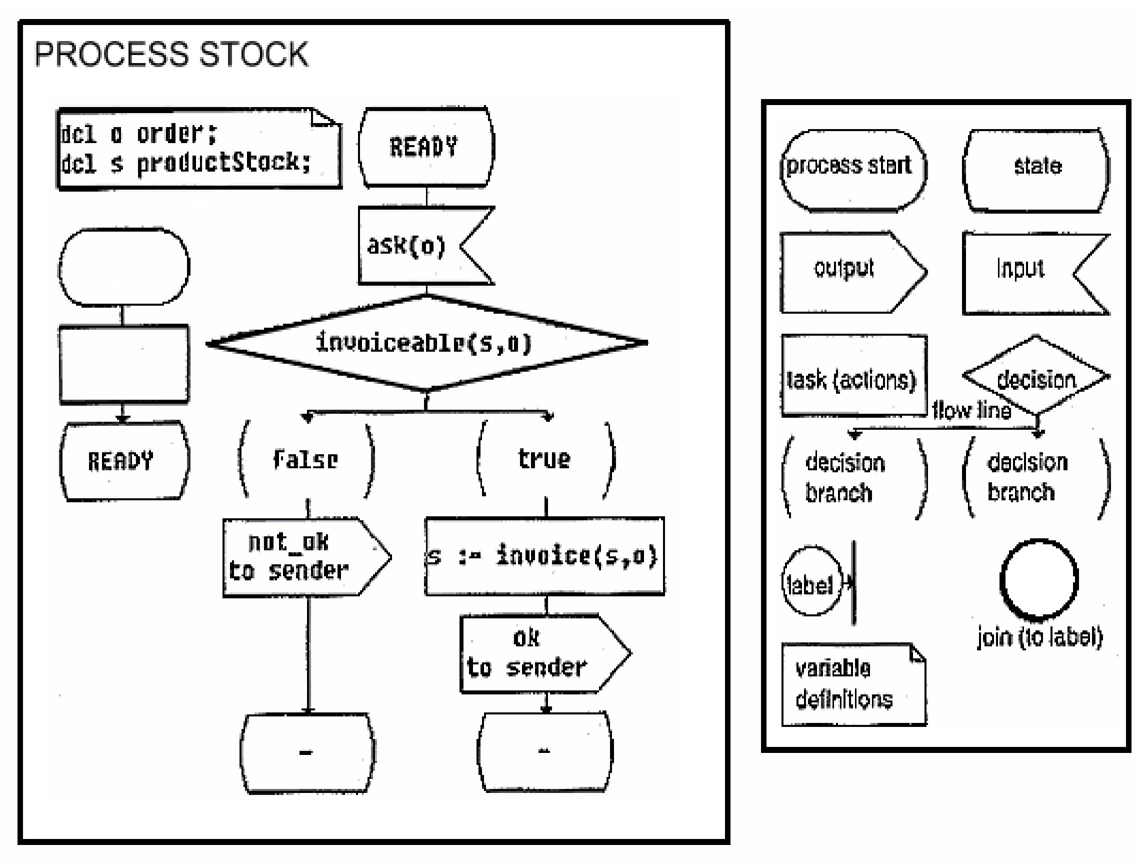


Figure 2 – Example SDL Specification and Key SDL Symbols

Figure 2 taken from [FRAPPIER] gives an impression of the widely used *Specification and Description Language* (*SDL*, see [SDL]), standardized by the INTERNATIONAL TELECOMMUNICATIONS UNION (*ITU*, see [ITU]). It is based on finite state machines running in parallel and communicating via “signals”. *ASN.1* (see [ASN1] and [ASN1BOOK]) is a present example for a notation used to describe data transmitted by dedicated protocols.

Although, unlike, e.g., STL or SDL the *Unified Modeling Language* (*UML*, see

[UML]) does not comprise formal semantic rules, various companies and organizations have been promoting it for many years as the industry-standard language for specifying, visualizing, constructing and documenting the artifacts of software systems. Currently attempts are made to unite the main concepts of SDL and UML (see, e.g., [REED]). Chapter 1 of [POSTON] contains more examples of specification languages.

Beside specifications of software requirements, also standardized methods to notate test cases have been evolved. A test case includes an unambiguous description, preferable in a format readably by software tools, of how an actual test shall be executed. We also speak of test scripts. Version three of the *Testing and Test Control Notation* (*TTCN-3*, see [TTCN3]) is currently one of the most popular test scripts languages. The significantly increased flexibility in its newest version allows the usage in many other sectors than the original one, the telecommunication. The most important language constructs are so-called components which communicate with each other via so-called ports. See [BAUMG] for a comprehensive introduction. Software companies like TELELOGIC (see [TELELOGIC]) offer editor applications for TTCN.

2.5 Test Automation

The main reason for automating the test process is to reduce testing errors, as well as, testing costs. Also, as stated in [GRAHAM2], leading companies have already achieved reductions in testing time of up to 70%, and 30% improvements in software development productivity by using automation tools.

Human beings are error prone in the things they do, which is just a consequence of being human, and so software testing done by humans is in principle error prone, too. Automating testing not only reduces the number of errors because it (partly) removes the human “component” with its individual differences concerning the tester’s education, training and experience as well as his work habits. It allows testing to occur as the code is written, which reduces the developers' tendency to sacrifice quality for productivity. And, moreover, the testing process can be consecutively improved over multiple uses.

Another advantage is the fact that automating testing captures knowledge that is ordinarily kept in the tester's brain and which would go to depart with him as soon as he starts working for a different company, as happens quite often during a tester’s career. If the testing process is documented and implemented via a testing tool, the knowledge stays in the test repository, and even a new tester can come up to speed with a bit of training on the tool and can understand how the software was tested in the past.

As is the case with software development, the most intense costs associated with software testing are costs of the humans who do the testing. Although designing and constructing test cases with an automated testing tool does require an initial investment of person hours up front, the overall human effort has been shown to be reduced by 50% (see [VTEST]).

Despite all expectable advantages of test automation which have been discussed and researched since the early 1980’s, in many software companies it is still not a mature process. And D.R. Graham’s statement, made in 1990, still applies:

“Testing has often been perceived as a tedious activity, yet it is seldom adequately tool-supported.” (in [GRAHAM]).

This is not a problem of unavailability of testing tools. See, e.g., the appendix of [KIT] and Chapter 18 in [PERRY] for extensive tool lists. The fact that companies usually have very specific test strategies makes the use of such, mostly quite universal, applications difficult, and often proprietary tools are developed instead. To meet the individual requirements suppliers of automation tools increasingly offer dedicated adaptations.

What can be automated during the test process? R.M. Poston distinguishes in [POSTON] between three phases: test case generation, execution and evaluation. Generation is the phase supported by the fewest number of today’s tools, probably because it is the most complex. Test cases that have been created according to intelligent guidelines enforced by a tool and stored in the tool's test repository are expected to be not random in nature as most manually constructed test cases tend to be. They should not leave testing gaps or redundantly test the same sections of the code. Subchapter 2.6 deals more intensively with this topic and in Chapters 4 and 5 the author’s approach is described in detail.

Concerning the second phase, also called the test run, many applications are available executing predefined test cases. In this way also the simulation of stress situations, e.g., by providing user input in a speed which could not be reached manually, is possible. But especially for testing *GUI*-interfaces there is still a lack of suitable tools, although some promising attempts exist (see, e.g., [APTEST])

In the final phase, the test case evaluation, the actual test results are compared with the expected ones to discover if software passed or failed a specific case. This can be supported by tools generating test oracles or taking the initial data states into account. Chapter 4 in [POSTON] intensively examines this topic. After all, the creation of detailed statistics about the test results is a very important functionality expected from good evaluation software. Furthermore, various coverage tools exist which can be used to receive particular information about, for example, memory consumption, function call coverage or *CPU* load during the test session.

Is test automation the ultimate solution? This question is discussed very controversially and, e.g., the authors of [STOCKS] believe that totally automating the testing process may be impossible. It is also not desirable, since a human tester can bring much insight to testing, as well as a degree of experience and wisdom in test case selection. Graham, moreover, states in [GRAHAM] that manual testing can find more errors than current software tools. A software tool can find all occurrences of some types of error, but cannot find all types of error, only the types which it is capable of looking for. Most tools currently available offer assistance in detecting only syntactic errors, not semantic errors.

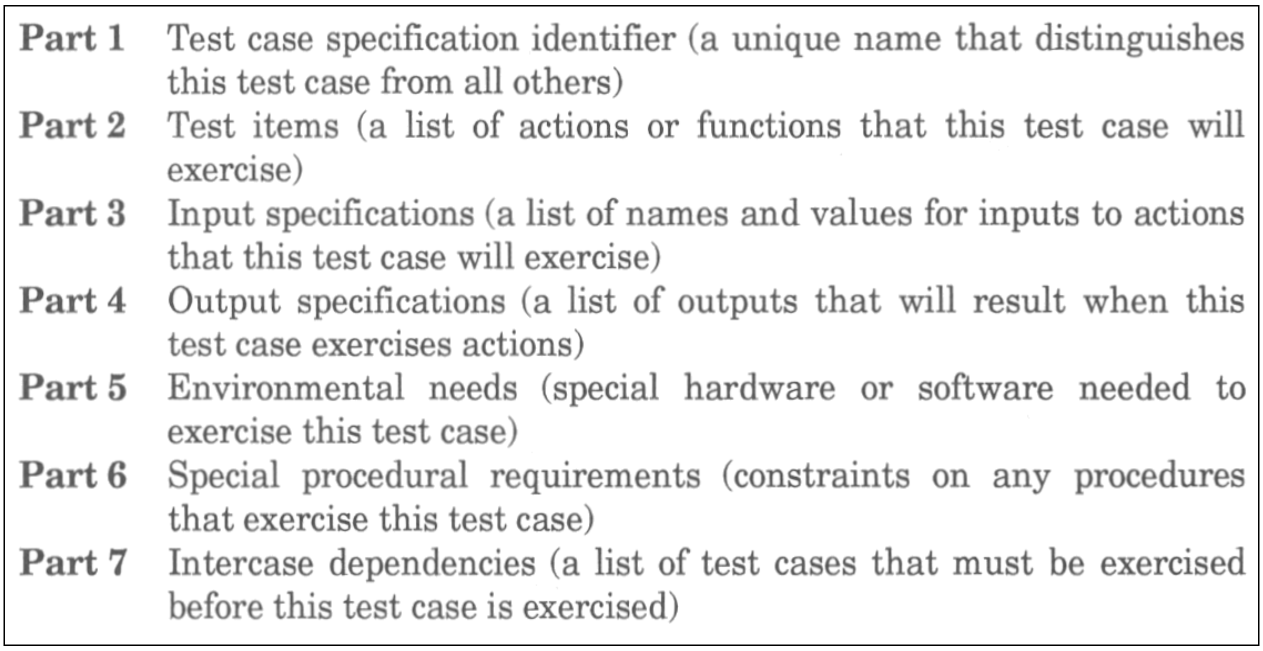
See [AUTOMATION] for detailed disquisitions about test automation.

2.6 Test Case Generators

The advantages expected from automated test case generation are increased speed during creation of new cases, but also more specialized and less redundant scenarios which should, nevertheless, cover all parts of code. While the first point seems quite obvious the second is the actual challenge.

J.B. Goodenough and S.L. Gerhart proposed in [GOODEN] a theory of test data selection which provides a basis for constructing program tests. This theory defines test data selection criteria in terms of properties called validity and reliability. They try to prove a fundamental theorem stating that successful execution of test data satisfying a valid and reliable selection criterion guarantees absence of errors in a program. Concerning this theorem testing can show the absence of errors, but only when the tests are properly selected. The author is not aware of generator software achieving that goal.

As basic precondition an automated generator tool needs a precise definition of what a test case means in a given context. *ANSI*/IEEE Standard 829 (see [IEEE91]) defines a test case specification as a document consisting of seven parts, as shown in Figure 3.



Part 1	Test case specification identifier (a unique name that distinguishes this test case from all others)
Part 2	Test items (a list of actions or functions that this test case will exercise)
Part 3	Input specifications (a list of names and values for inputs to actions that this test case will exercise)
Part 4	Output specifications (a list of outputs that will result when this test case exercises actions)
Part 5	Environmental needs (special hardware or software needed to exercise this test case)
Part 6	Special procedural requirements (constraints on any procedures that exercise this test case)
Part 7	Intercase dependencies (a list of test cases that must be exercised before this test case is exercised)

Figure 3 – Parts of a Test Case Specification

While Parts 5-7 usually have to be created only once for many cases and Part 1 and 2 can be generated quite easily, the most problematic work has to be done for Parts 3 and 4.

Where from can a generator tool know about input data and expected output? R.M. Poston differentiates in [POSTON] between three types of sources: specifications, program source code and test design languages. A.A. Omar and F.A. Mohammad distinguish concerning functional and structural testing ([OMAR89]). Functional testing involves the generation of test cases that are based on the requirements, specifications, and design functions of a program; structural testing makes use of the program structure in designing an adequate test case ([OMAR91]). With reference to [MYERS] the recommended procedure is to develop test cases using the functional method and then develop supplementary test cases as necessary by using the structural method.

G.J. Meyers, furthermore, states that the key issue in designing effective test cases is to yield the best subset of all the possible test cases, taking into consideration economic constraints, such as time and cost, which has the highest probability of detecting the most errors.

[POSTON] contains a detailed examination of how test case generation can be done based on specifications. Commercial tools exist for this method; see, for example, [TELELOGIC].

Torx, Autolink, TGV and UIO are examples for test derivation algorithms currently favored. A comparison between them regarding their possibilities and limits can be found in [REED].

3 Software and Testing in Cellular Networks

After the explanation of universal software testing procedures and automation methods in the previous subchapters the particular subject of this paper – software in the mobile sector – will now be examined.

Preceded by a general overview, the possibilities of error analysis together with tracing are described in detail, importance and status quo of automation are examined and finally a specific test script language will be introduced.

3.1 Overview

Concerning software for the mobile business, a large spectrum of issues has to be mentioned: server software at the base stations; software running on *routers*; several protocol layers inside, e.g., a cellular phone implementing the communication; and finally the end user applications, the so-called *Man Machine Interface* (*MMI*). The latter are of particular importance as already 75% of the inhabitants of Germany are confronted with them day by day – while searching the address book, writing a *SMS*, playing a game, etc.

This analysis will focus on the implementation of the communication part in mobile devices. In Europe the *ETSI* organization (see [ETSI]) provides official specifications of so-called *entities* and the control flow requested between them (via so-called *SAP*s) relative to the technique used, e.g., *GSM*, *GPRS* or *UMTS* (see [GSM], [GPRS] and [UMTS]). With respect to the OSI reference model of the *ISO* (see [ISOOSI]), these entities provide functionality for the network and the link layer. Together with lower hardware drivers and higher protocols, they make up the so-called *protocol stack* (PS, see Figure 4).

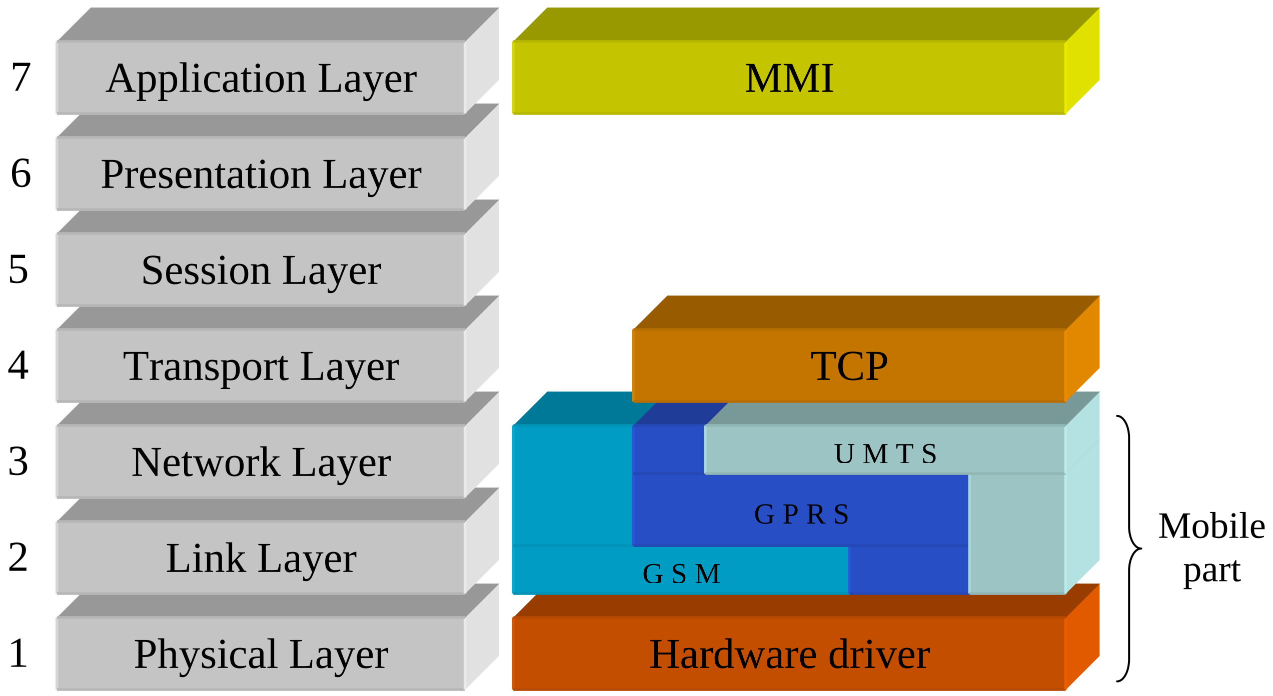


Figure 4 – ISO-OSI Reference Model Mapped to Mobile Device PS

Since no actual implementational details are specified by the ETSI, companies use various approaches. One of them is to realize the entities as separated tasks and build a framework which allows, e.g., the exchange of data packages (*primitives*) between them. Primitives are hierarchical structured as Table 3 shows exemplarily:

Element					**Type**
`cell_info`					`struct`
	`cell_env`				`struct`
		`Rai`			`struct`
			`plmn`		`struct`
				`v_plmn`	`ubyte`
				`mcc`	`ubyte`
				`mnc`	`ubyte`
			`lac`		`ushort`
			`rac`		`ubyte`
		`cid`			`ushort`
	`service_state`				`ubyte`
	`net_mode`				`ubyte`

Table 3 – Structure of the Primitive GMMRR_CELL_IND

3.2 Error Analysis in the Mobile Sector

At Texas Instruments analyzing software designed to run in a mobile device is done in three states: *Simulation-Test*, *Target-Test* and *Official Approval*.

Simulation-Test: As with “common” software, also in the mobile sector a first attempt to eliminate programming errors is done by running many function and module tests (see Subchapter 2.3). Typically this means that developers are checking and debugging the functionalities of their entities independently of extraneous influences – by, e.g., writing special test programs which call the new functions with varying parameters.

Even integration tests are possible without real mobile hardware or a network. To achieve that the protocol stack (containing the entities under test) will be compiled for the operating system running on the test PC (e.g., MS-WINDOWS) and linked to an emulation of the operating system which will be used on the hardware (e.g., NUCLEUS) later. Afterwards special software called the *TAP* (Test Application Process) is used to execute precompiled test cases in a defined order. In brief, data packages will be sent to selected entities and the resulting data are received and evaluated (see also Subchapter 4.5). This technique is a mixture of black- and white-box testing since it is possible to follow the control flow in a debugger; but mostly only the data sent out of the protocol stack simulation are visible. At this point *traces* are used for the first time. These are special data packages containing different information about the current state of the entities running inside the PS.

Target-Test: During the next test phase the protocol stack is actually loaded into an electronic board containing the same hardware as will the final product. Figure 5 gives an impression of a test board typically used in 2003. At this state it is possible to detect software failures not occurring in the PC emulation.

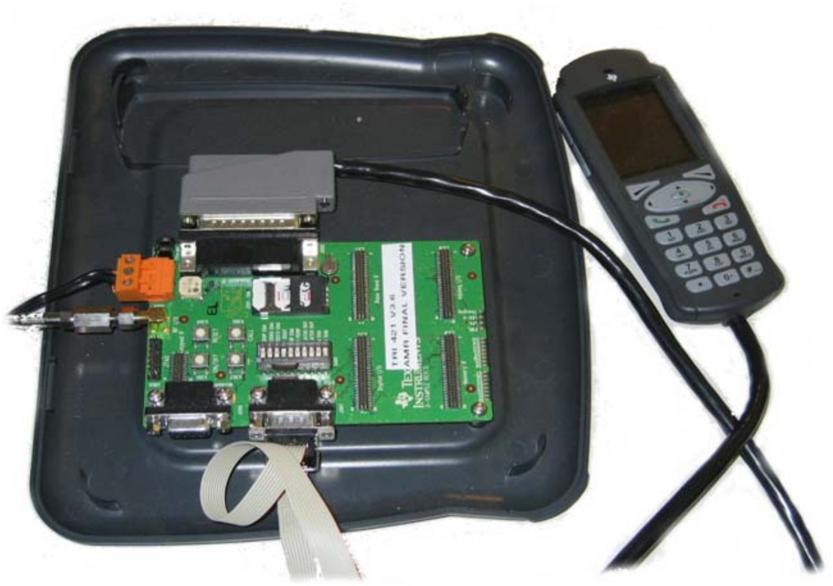


Figure 5 – Electronic Board "D-Sample"

The board usually provides an interface (e.g., serial or *USB* ports) over which data may be exchanged. That can be *AT* commands (international standardized strings in *ASCII* format) or proprietary messages like the mentioned traces (see Subchapter 4.1 for more details). During so-called *field tests* testers drive with such a test board across the country and run several experiments, in fact try to check out all the capabilities of the product. But also inside the labs test boards are used for several purposes like

– *Quick Tests*, typically run to check the major functions (e.g., setting up calls, *SMS* sending or data transmission) after small software changes
– or *Robustness Tests* where, e.g., the board is just switched on and after a week is checked whether it is still functioning.

During all these tests the produced traces provide information about progress and success. In addition they are recorded for further analysis and error reproduction in the PC emulation. Although in connection with special hardware tools it is also possible to directly debug the software *image-file* on board, this possibility is currently seldom used due to its intricateness. In fact traces are mostly the only way for developers to understand an error prone behavior.

Official Approval: This last category is very important concerning the acceptance by the fabricators of mobile devices. The ETSI as well as the different *network operators* (e.g., E-PLUS) specify sets of test cases and the runtime conditions. Protocol stack software not passing these tests will be considered to be not ETSI-conform, which discourages potential costumers, and furthermore it will most likely not function under realistic conditions. Officially the tests are referred to as *FTA*s and *IOT*s as described below:

- FTA – *Final Test Approval*: FTA specifications are formulated by the ETSI and quite many companies offering testing capabilities (like ANITE ([ANITE]), AGILENT ([AGILENT]) or RHODE&SCHWARZ ([RSD])), provide advanced test systems (e.g., *SAT – Stand Alone Tester* from ANITE) together with dedicated software which can be used to run the test cases under basically simulated air network conditions. The term “basically simulated” is important since, for example, unlike in real networks a data package not processed at the first attempt won’t be retransmitted.

  So-called test houses (laboratories maintained by such companies) may be used instead of installing all the hard- and software in a private test lab. TEXAS INSTRUMENTS uses both possibilities.
- IOT – *In-Orbit Test*: As the name implies these tests are mostly run within a real network using special base stations provided by the network operators, although the latter sometimes also use simulators (e.g., *SAS – Stand Alone Simulator*). But these are actually emulating a network – with, e.g., all kinds of transmission problems. Each network operator has its own set of test specifications.

All in all, the Official Approval consists of extensive integration tests under almost realistic conditions. Traces are very important in this context since mostly no hardware debugging facilities exist in test houses or at sites of network operators, and every problem has to be reproduced later.

As a conclusion, it can be stated that testing in the mobile sector mainly consists of black-box procedures executed by independent testers where the results will be used by the actual developers to fix bugs and solve problems. Obviously these results have to be actually meaningful and so we come to the requirements demanded by developers and testers.

3.3 Importance of Automation

As described in 2.5 automation of the test processes helps reducing testing errors, as well as, testing costs. Especially in the mobile sector, where new requirements are upcoming every month and therefore new features and functionalities are permanently introduced in the software, testing needs to be very fast but still efficient.

Particularly regression testing is done daily and, if done manually, would require very patient testers executing almost the same procedure day after day – with increasing tendency to make mistakes. Long-term tests including waiting phases of many ours are another example of typical analyses in the mobile business which no human could endure for long. An alternative would be to magnificently increase the count of testers, leading of course to unnecessarily high payroll costs.

Since most of today's tests of mobile software are so complex that tools are needed for their execution anyway, the automated run of such tools is the main goal pursued in these days. And already a lot of commercial applications exist for this purpose, like the Stand Alone Tester mentioned above. The SAT consists of a hard-/software combination which is, e.g., able to simulate air-network conditions according to test cases written in a proprietary script language.

At TEXAS INSTRUMENTS moreover many internally developed tools exist, which are widely used during the various test scenarios. As pointed out, tracing mechanisms play a central roll – and so the PCO tool (see Subchapter 4.3 for more details and [TRACING] for a complete disquisition). It most of all supports visualization of status information about the active entities. Such is provided via traces – special data packages sent out of the protocol stack. As mentioned before, this can be done via a serial cable. If the protocol stack is running on the same computer shared memory is another option (further possibilities like network communication are not considered in this paper). Precondition is an instrumentation of the entity source code, e.g., at the entry points of functions or after significant changes of the internal state. Beside such status messages monitoring the content of the actual primitives exchanged between the protocol stack entities is supported. To make this possible complete information about the used primitives and their structures is needed. A possibility to manage data like that is explained in Subchapter 4.2.

To stimulate a protocol stack tools like *xPanel* (also described in [TRACING]) and TAP (see Subchapter 4.5) are used. The first is able to send key presses – via a GUI-interface as well as from the command line, which is useful to automate MMI tests where otherwise a tester would have to manually input key sequences.

The TAP is able to send (and receive) any kind of primitives to (and from) the protocol stack under test. In fact by executing test cases written in the proprietary script language TDC (see Subchapter 3.4) it simulates all entities communicating with the *entities under test* (*EUT*) and verifies the correct behavior of the latter. Such tests typically run automatically, even at night time. But together with a software/hardware debugger the developers can also use them to reproduce and directly analyze errors.

While the execution of test cases is highly automated, computerized generation of new ones is still not very common. In fact, at TI-Berlin all test cases have been written manually till now. That involves a costly and time consuming process where developers, familiar with the specific protocol stack internals, have to think about possible situations and sequences which should be tested. Moreover, already existing test cases have to be regularly updated – a big problem if done completely manually. Of course, several attempts where made to introduce more automatisms, but the examined commercial tools (like CMICRO from [TELELOGIC]) mostly expected a whole proprietary framework and the effort to combine this with the existing was always judged to be unacceptable.

The fact that together with the mentioned tracing a logging mechanism exists, which stores all received information as needed, lead to considerations on how to use this data for the test case creation. The topicality of the recorded information would be one obvious advance. Chapter 4 contains the theory of how test case generation using traced data can be performed. In chapter 5 an actual implementation is described.

3.4 Specifications

In the telecommunication sector the necessity to have manufacturer-independent and precise standards for communication protocols and services was realized quite from the beginning, unlike in many other segments of computer science. The ITU standardized SDL and *Message Sequence Charts* (*MSC*, see [MSC]), and recommendations by the ETSI are available in this formats.

MSC is a graphical specification language by which the communication behavior, e.g., between entities inside a protocol stack can be described. Figure 6 contains an example chart for the SAPs between entities MMI, *MM*, *GMM* and *GRR*; it represents the process of enabling the GPRS capability of a mobile. At TEXAS INSTRUMENTS in Berlin such specifications are used by developers as a reference during the implementation process. They work with a visualization tool from CINDERELLA (see [CINDERELLA]).

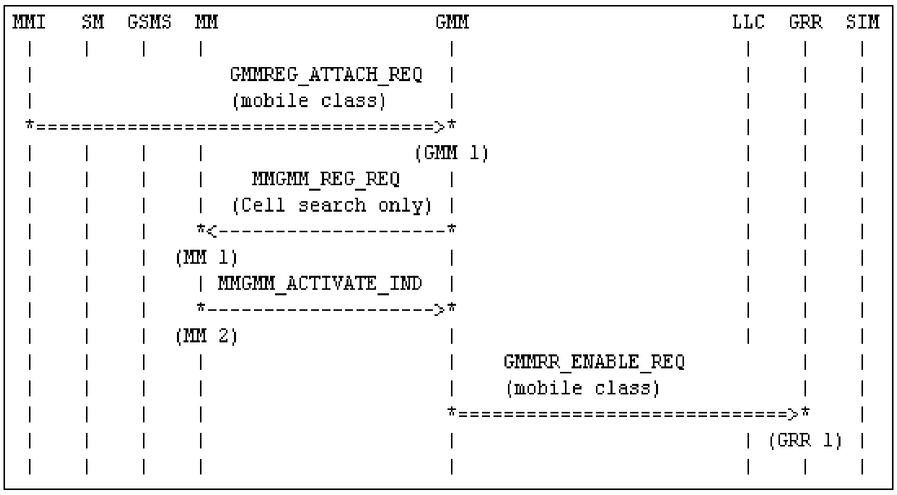


Figure 6 – Example Message Sequence Chart

But usually not the original ETSI specs are used. According to them definitions of SAPs and air message interfaces are maintained in an internal format, which can also be understood by dedicated tools developed in Berlin for generation of, e.g., header files and information databases accessible by the protocol stack software. See Subchapter 5.3, section **CCD-Database**, for further details.

Regarding the testing process MSCs are used again as reference during writing of test cases. Furthermore, tools exist to reconstruct sequence charts of the actual message flow as it occurred during a test. The mentioned information databases also take an important role for testing.

3.5 TDC - Test Description Code

The *Test Description Code* (*TDC*, see also [TDC]) is the proprietary language currently used at TEXAS INSTRUMENTS to describe test cases. It provides a subset of TTCN-3 features and has been introduced as an intermediate solution on the way to the latter standardized language. Basically, primitives with certain content are defined and, via `SEND/AWAIT` commands, transmitted.

The TDC syntax consists of a mixture of pure *C++* (see [CPP]) statements and predefined C++ templates. Therefore, a TDC test case is directly compilable by a standard C++ compiler. Beside the fact that C++ is a well known language by most developers/testers, furthermore, the modularization capabilities of the language can be used and the *code completion* features in up-to-date development environments help to speed up test case editing.

Together with the TAP (see Subchapter 4.5), which executes TDC cases, exception handling via `TRAP` and `ONFAIL` statements and so-called “parking” of primitives are provided. The latter comes in useful if the chronological order of data transmitted is allowed to vary to a certain degree.

Figure 7 contains the partial *BNF* for the TDC syntax. Italic key words are defined either in the BNF for C or in further TDC specs (see also Subchapter 5.5.2). The distribution into multiple header/source files is not considered here. An example of an actual TDC test case can be found in Appendix A.1.

```
<Testcase>               ::= ‘T_CASE’ <Name> ‘( ) { BEGIN_CASE (‘ <Comment> ‘) {‘
                             <TestcaseBody> ‘} }’
<TestcaseBody>           ::= [<TeststepList>]
<TeststepList>           ::= <Teststep> [<TeststepList>]
<Teststep>               ::= ‘T_STEP’ <Name> ‘( ) { BEGIN_STEP (‘ <Comment> ‘) {‘
                             <TeststepBody> ‘} }’
<TeststepBody>           ::= [<StatementList>]
<StatementList>          ::= <Statement> [<StatementList>]
<Statement>              ::= <SendStatement> | <AwaitStatement> | <CommandStatement> |
                             <TimeoutStatement> | <FailStatement> | <PassStatement> |
                             <AlternativeStatement> | <TrapStatement> |
                             <CcodeStatement> |‘{‘ [<StatementList>] ‘}’
<SendStatement>          ::= ‘SEND’ ‘(‘ <Primitive> ‘)’ ‘;’
<AwaitStatement>         ::= ‘AWAIT’ ‘(‘ <Primitive> ‘)’ ‘;’
<Primitive>              ::= ‘T_PRIMITIVE_UNION’ <Name> ‘(‘ [<Arguments>] ‘) {’
                             <C struct assignment> ‘}’
<CommandStatement>       ::= ‘COMMAND’ ‘(‘ <CommandString> ‘)’ ‘;’
<TimeoutStatement>       ::= ‘TIMEOUT’ ‘(‘ <Milliseconds> ‘)’ ‘;’ |
                             ‘TIMEOUT_WAIT’ ‘(‘ <Milliseconds> ‘)’ ‘;’ |
                             ‘MUTE’ ‘(‘ <Milliseconds> ‘)’ ‘;’ |
<FailStatement>          ::= ‘FAIL’ ‘(‘ ‘)’ ‘;’
<PassStatement>          ::= ‘PASS’ ‘(‘ ‘)’ ‘;’
<AlternativeStatement>   ::= ‘ALT’ ‘{‘ <OnStatementList> <OtherwiseStatement> ‘}’
<OnStatementList>        ::= <OnStatement> [<OnStatementList>]
<OnStatement>            ::= ‘ON’ ‘(‘ <AwaitStatement> ‘)’ <Statement>
<OtherwiseStatement>     ::= ‘OTHERWISE’ ‘(‘ ‘)’ <Statement>
<TrapStatement>          ::= ‘TRAP’ <Statement> ‘ONFAIL’ <Statement>
<CcodeStatement>         ::= ‘if’ ‘(‘ ‘)’ ‘{‘ ‘}’ ‘else’  ‘{‘ ‘}’ |
                             ‘for’ ‘(‘ ‘)’ ‘{‘ ‘}’  | ‘while’ ‘(‘ ‘)’ ‘{‘ ‘}’ ‘;’  |
                             ‘do’ ‘{‘ ‘}’ ‘while’ ‘(‘ ‘)’ ‘;’  |
                             ‘switch’ ‘(‘ ‘)’ ‘{ ‘ ‘case’ ‘:’ ‘}’ ‘;’ |
                             <C variable declaration>
<CommandString>          ::= <FromEntity> ‘REDIRECT’ <Orig_Dest_Entity> <New_Dest_Entity> |
                             ‘RESET’ <Entity> |
                             <FromEntity> ‘DUPLICATE’ <Orig_Dest_Entity> <New_Dest_Entity> |
                             <Entity> ‘CONFIG’ <ConfigString>
<Milliseconds>           ::= <C integer>
<FromEntity>             ::= <C string>
<Orig_Dest_Entity>       ::= <C string>
<New_Dest_Entity>        ::= <C string>
<Name>                   ::= <C string>
<ConfigString>           ::= <C string>
<Comment>                ::= <C string>
```

Figure 7 – Partial BNF of the TDC Language

4 The Test Case Generator – Theoretical Concepts

In addition to the concepts of the test case generator (*TCGen*) implemented together with this thesis, this chapter contains theoretical basics concerning the testing environment as used in the TI laboratories in Berlin – with the main focus on the previously mentioned tracing and its usage for test case generation.

After an introduction in the general idea for the generator, the framework conditions will be described in Subchapter 4.2. The subsequent sections contain explanations of the generation process and other test applications involved.

4.1 General Idea

As described in Subchapters 3.2 and 3.3, simulation tests take an important role during the development process of mobile software; writing new or updating existing test cases is a critical procedure often neglected due to a lack of automation. On the other hand, large amounts of logged data are created during field tests. This data is mainly used to review erroneous behavior by replaying them offline. But in the case of succeeded tests the logfiles contain primitive/message flows as requested by the specification. A generator accepting them as input could produce up-to-date test cases.

Of course, such generated tests would only examine already tested functionalities, but regression tests are the type of software checks applied most often and which run, typically, highly automated. Moreover, as explained at greater detail in Section 4.4.2, various rules have to be applied during generation to produce correct cases. The rule processing could also be used to vary the behavior and by doing this create test routines for new functionality, as well. Later on, the generator might also be enhanced to accept, e.g., *MSC* specifications (see Subchapter 3.4) as further input. But even if test cases for new behaviors still have to be written manually, the generated cases would be a good basis for developers.

Another usage option for a test case generator as proposed is the possibility to exactly reproduce field test situations at a PC-simulation. This way no actual air network would be necessary to, e.g., analyze an error which appeared while sending an SMS. To do this the entities in which a bug is expected have to be separated from the rest of the protocol stack. Section 4.5 introduces techniques to achieve this.

A prerequisite for a meaningful generation process is the logging of all data necessary. See Subchapters 4.2 (Section **Entity Graph**) and 4.3.2 for suitable methods.

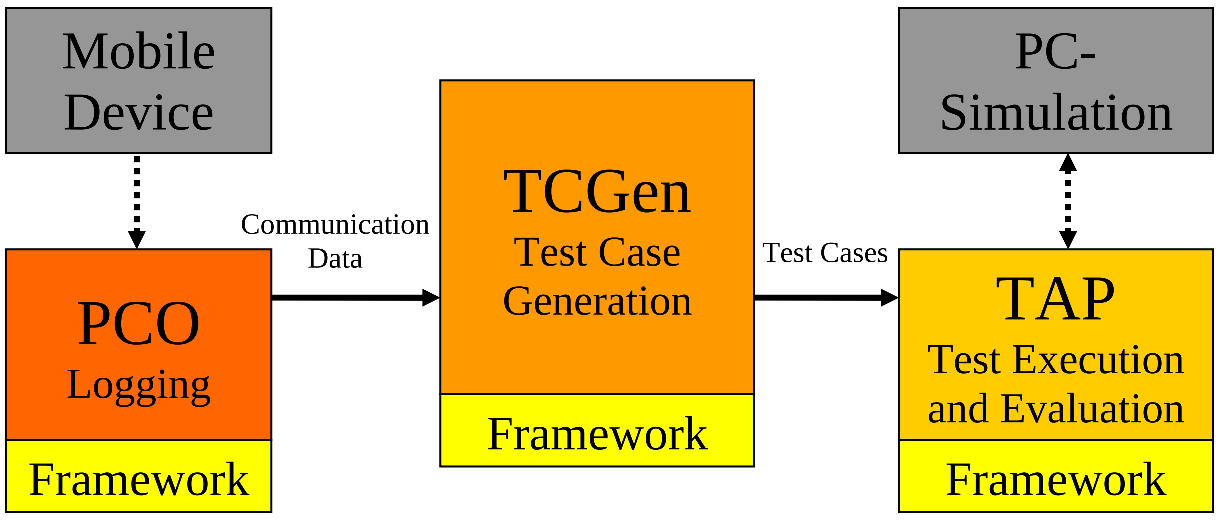


Figure 8 – Test Case Generation Tool Chain

In Figure 8 the existing test applications and the test case generator with their dependencies regarding the author’s approach are displayed. In brief, the tool PCO (see Subchapter 4.3) is responsible for the logging of communication data, the latter is sourced by the generator TCGen (see Subchapter 4.4) to create new test cases which then will be executed and evaluated by the tool TAP (see Subchapter 4.5). All the applications in this chain use functionality of the framework described in the next subchapter.

4.2 The Framework

Frameworks for software testing are primarily proprietary inventions of the individual company. At TEXAS INSTRUMENTS the *FRAME* (including a so-called *Test Interface*) is the basis for the whole protocol stack – but also for testing as described below.

Test Interface Approach: To test the behavior of a protocol stack it is necessary to watch the communication flow and state changes inside the running system. Using a debugger tool allows direct access to, e.g., content of variables or the call stack but this has destructive influence on time critical operations. And debugging actual mobile hardware requires expensive test hardware.

The Test Interface provides the possibility to route internal information out of the protocol stack. It consists of dedicated software pieces on both ends – protocol stack and test PC – which communicate via, e.g., a serial cable or shared memory. Communication in this context means stimulation of the protocol stack by sending primitives containing system commands to it and vice versa receiving traces and duplicated primitives. In this way, the PS can be observed without interrupting its interactivity with the radio network. Of course, only selected information should be routed out at a time to keep the Test Interface traffic as low as possible. Figure 9 gives an overview of this approach:

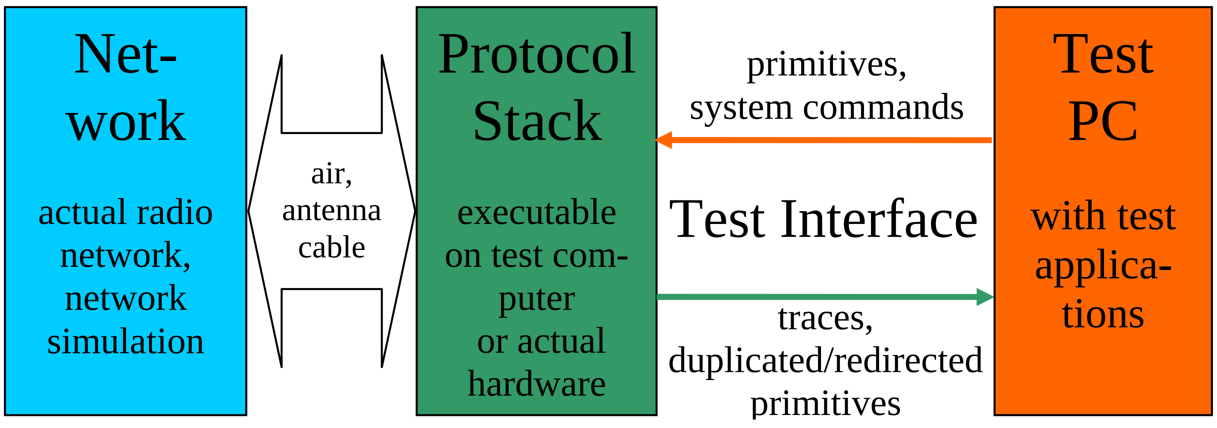

Net-work
actual radio network, network simulation
air, antenna cable
Protocol Stack
executable on test com-puter or actual hardware
primitives, system commands
Test Interface
traces, duplicated/redirected primitives
Test PC
with test applica-tions

Figure 9 – Test Interface Approach

The FRAME: The term *FRAME* stands for the whole runtime environment under which all entities are running. More precisely it provides a life cycle management for tasks running in parallel, and, in the context of a protocol stack, an entity is implemented by one or more tasks. Furthermore the FRAME contains functionality for synchronization, timer handling and memory management. A queuing mechanism also allows inter-task communication. This enables passing of primitives and, therefore, any kind of data between entities. It is, in fact, an abstraction from the operating system actually used.

The communication techniques of FRAME (see Figure 10) are of great importance for software testing. They make it possible to route primitives into and out of the protocol stack. While requesting the latter it can be decided whether the primitives should just be duplicated (without changing the communication flow) or actually redirected. Another possibility is to completely disable communication queues.

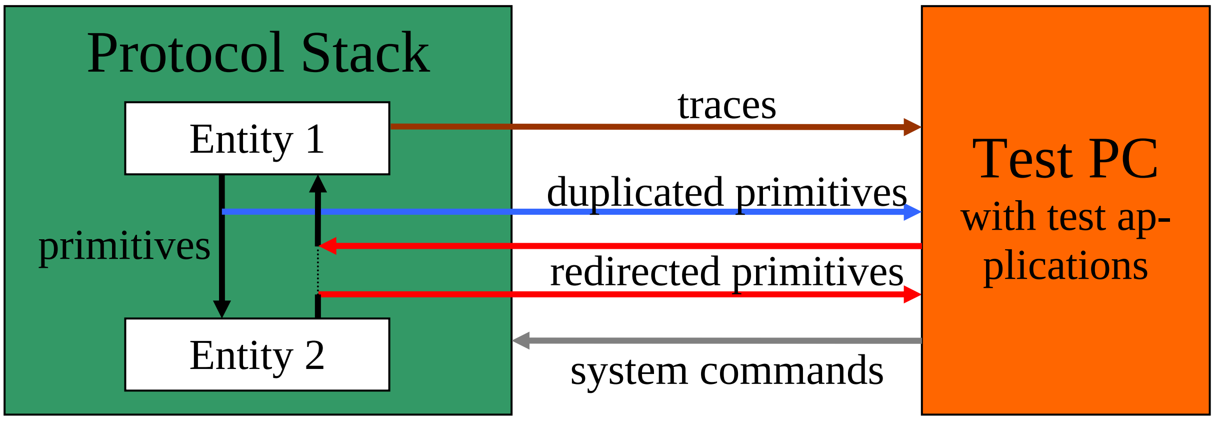


Figure 10 – Routing Possibilities of the FRAME

Traces and system commands are also just special primitives (or better packed inside them) and, therefore, are transported in the same way. In fact, they are ASCII-strings as described in more detail in Subchapter 5.3. Each trace is assigned to a *trace-class*, e.g., all error traces to class "error". Commands exist to enable/disable the production of traces belonging to a specific class per entity.

The actual routing settings can also be selected by sending a dedicated system command to the protocol stack. The FRAME running there will acknowledge by returning one or more traces (e.g., “OK”). This is surely not the best solution. If, for example, automatic handling of such communication is needed every trace has to be parsed. Special response commands would be a good alternative.

CCD-Database: *CCD* is the abbreviation for the *Condat-Coder/Decoder* which was originally developed by the company CONDAT AG. It is a software library mainly used to code and decode *air messages*, according to various official communication standards like ASN.1 ([ASN1]). Air messages are the data blocks finally sent via the cellular network. Inside a mobile device they are carried within primitives.

The so-called CCD-Database contains information about the structure of all air message types used and also about the primitives. Due to the fact that this database is also available to test applications, e.g., duplicated primitives – in fact a byte stream delivered to the test PC – can be reconstructed and represented to testers with all details like values of certain parameters.

Entity Graph: Figure 11 contains a graphic overview of the relationship between entities in a protocol stack regarding their communication with each other. These relations can be considered as a finite and directed multi-*graph* $EG_1(V, S)$ where the set of vertices (V) corresponds to the set of entities existing in a specific PS and S represents the Service Access Points. A function $w_1: S \rightarrow \mathcal{P}(V) \times \mathcal{P}(V)$ assigns each SAP a set of entities using the functionalities of one or more other entities, for example, *(*{GMM*, GSMS, SNDCP}, {LLC})* to the *LL*-SAP or *({ACI},{*SMS,GSMS*})* to the *MNSMS*-SAP. A simpler representation as undirected graph $EG_2(V, E)$ construes the edges (E) as primitive interfaces between two entities. The corresponding assignment function is $w_2: E \rightarrow I$ with $I \subset \mathcal{P}(V) \wedge \forall i \in I(|i|=2)$. See [GRAPH] for an introduction in graph theory.

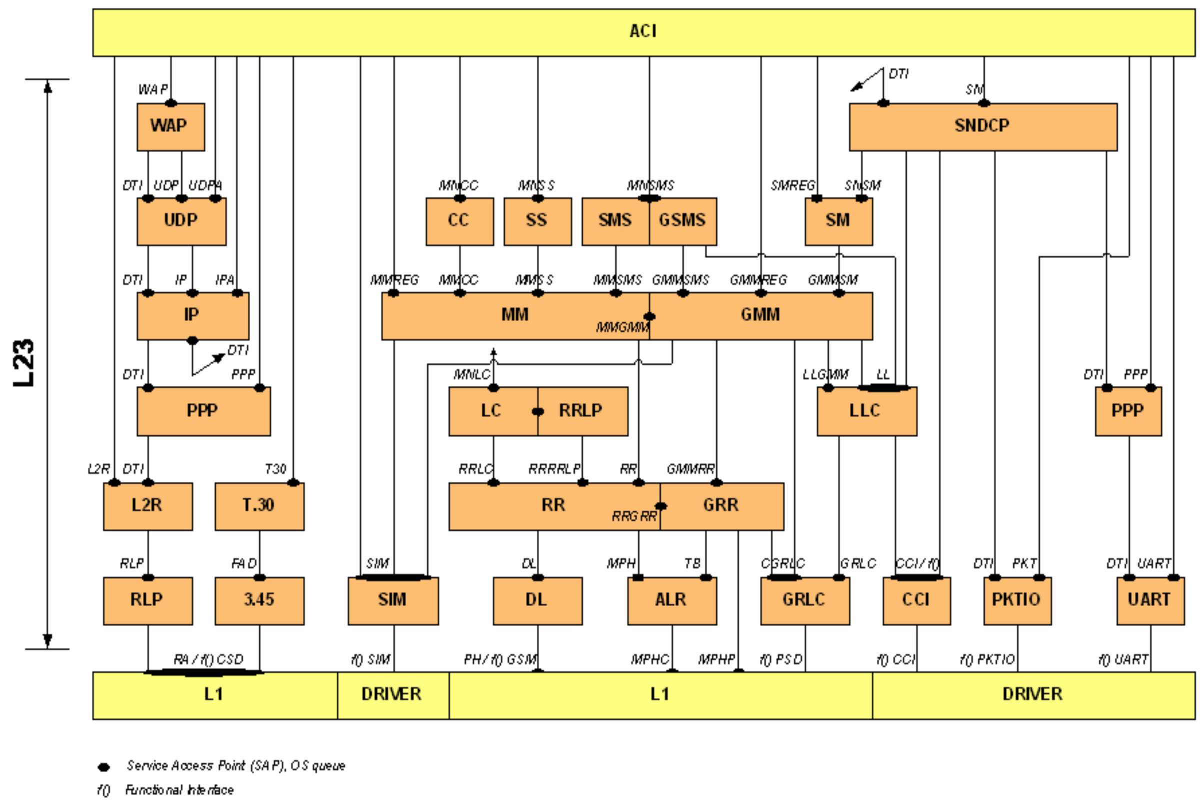


Figure 11 – Relations between Entities of a GPRS-PS

By maintaining such a graph and providing various access functions, the framework enables test tools to easily find out which entities communicate with each other.

When speaking of the framework in context of the protocol stack as well as of test applications, the same concepts are referred to. In fact FRAME and CCD are generic and, therefore, available for mobile operating systems as well as personal computers (see Subchapter 5.3). Tools like TAP running on a test PC can use the whole task management of FRAME.

Other test software might be useful for very different needs and should not depend that much on a specific runtime system. Thus, the applications developed by the author, PCO and TCGen, utilize only specific parts of the framework, for example the routing functionality, as described in the next sections. Refer to Subchapters 5.4 and 5.5 for a description of actual implementations.

4.3 PCO – Point of Control and Observation

The test tool PCO is mainly used for monitoring the current state of a protocol stack without changing it. PCO can be combined with more active tools like TAP (see Subchapter 4.5) or xPanel (see [TRACING]).

As the term “Control” in the name implies, it is not an exclusively passive application. For instance, it provides the tester with an interface to decide what should be monitored. This user input is then realized by sending appropriate system commands to the protocol stack. In return, traces and duplicated primitives can be received and displayed. Figure 12 provides an overview about the general functionality:

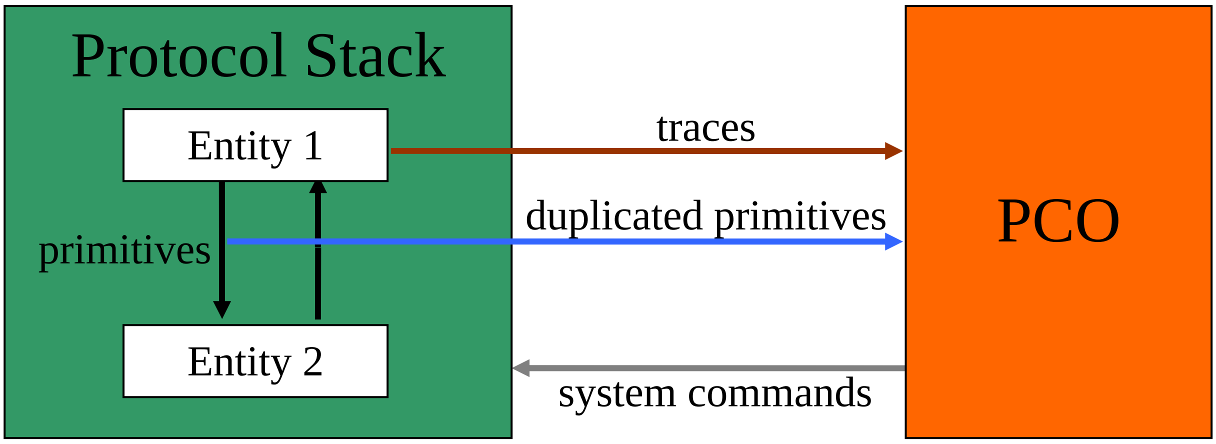


Figure 12 – PCO Communications

Visualizing traces is quite easy since only ASCII-strings have to be displayed. However, as described in Section 5.4.2, much can be done to make the user interface convenient. To present duplicated protocol stack primitives in a meaningful way, it is not sufficient to show the received data as hexadecimal dump – although this is also supported. The tester/developer wants to look inside the structure of each primitive, see the current value of each parameter and even have air messages decoded. PCO provides all this using the CCD-Database. This requires a database matching the protocol stack actually used, which has to be supplied by the user of PCO.

Aside from supervising the situation inside the PS during a test, another main feature of the PCO tool is its capability of logging test sessions: all data arriving via the Test Interface can be stored and replayed if needed.

4.3.1 *The Universal Viewer Concept*

Although in the previous descriptions PCO has been referred to as one application, it actually consists of three conceptual components which are communicating with each other (see Figure 13).

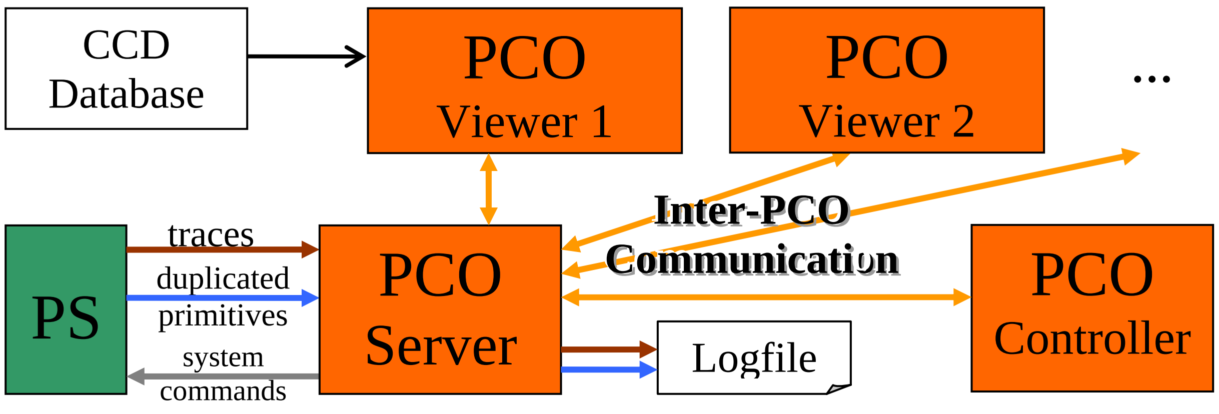


Figure 13 – PCO Components

The *PCO-Server* is the central part where all trace or primitive data arrive and from where they are distributed. It is directly connected to the Test Interface.

The main function of the *PCO-Controller* is to provide an interface to the server. They communicate by using a dedicated *PCO-protocol*. This way, e.g., the logging process can be started/stopped, system commands are requested to be sent to the protocol stack and Test Interface communication parameters (like the COM port for a serial connection) can be specified. A controller implementation may also supervise a whole test environment consisting of the PCO parts and other applications like xPanel.

The third PCO component type, the *PCO-Viewer*, is responsible for the visualization of received information and data. Because this can be done in quite different ways depending on specific needs, a universal concept has been worked out by the author. It basically consists of a so-called *Viewer-Interface* whose actual implementational details can be found in Subchapter 5.4.3. Any application supporting this interface can connect to the PCO-server and request data from a currently running PS or from a PCO-logfile, matching individual filter settings. The server does not differentiate between the actual viewer implementations but communicates with all of them via the same PCO-protocol messages. Data is provided together with information about the sending and the receiving entity, as well as with a timestamp. A viewer might, e.g., present traces and primitives as a visual list to the user and use the CCD-Database to interpret/decode hex dumps. It could as well forward the data to a database server for later processing. In fact, users can easily create their own special viewer and use all existing capabilities of PCO and the framework.

The PCO-protocol also supports messages for synchronization between different viewers, e.g., concerning time stamps. These messages are sent from one viewer to the server which forwards them to all other connected viewers.

4.3.2 Observation of Entities

To observe an entity, specific trace-classes can be enabled for it to retrieve information about internal states. Furthermore, the duplication of primitives sent and received by the entity should be requested. This can be done by forwarding appropriate system commands to FRAME running on protocol stack side. To do so the user must know about all primitive interfaces to other entities and provide this information to the PCO tool. But especially if he is interested in more then one entity this method is quite difficult and inconvenient.

PCO supports users by utilizing the Entity Graph provided by the framework. First of all, the graph enables PCO to provide testers with a list of all existing entities from which specific ones can be selected. Now duplication commands for all communication edges can be generated. But since often only the so-called outer primitive flow is of interest, meaning the communication with all observed entities considered as one block, the tool can also create a sub-graph by selecting all needed entity-nodes and combining vertices and edges accordingly.

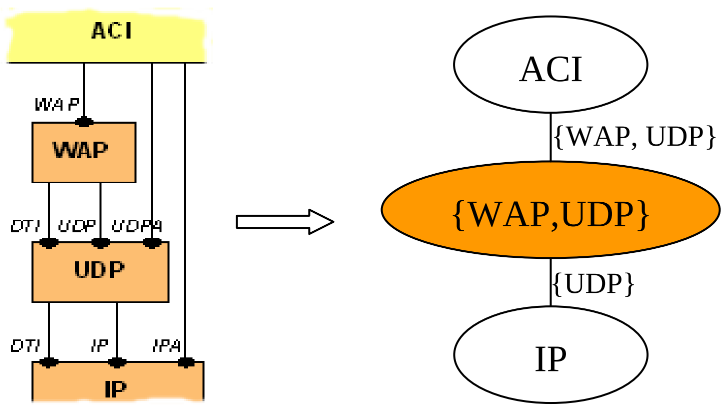


Figure 14 – Sub-graph Creation to Observe Two Entities

Figure 14 exemplarily demonstrates this functionality for the entities *WAP* and *UDP* which exchange primitives with ACI and *IP*. The Entity Graph shown in Figure 11 on page 34 has been taken as basis. Only duplication commands for edges of the new graph will now be sent. This is sufficient, because duplication settings can but do not have to depend on information about SAPs. To avoid unnecessary commands the connections in the sub-graph are enhanced by labels, each containing a subset of the observed entity-block representing the entities actually exchanging primitives with the particular outer one.

By additionally using the logging functionality of PCO input for the generator tool TCGen can be created. The observed entities will then later be the ones under test. In case it is not possible to decide beforehand which communication data will be needed, another possibility is to duplicate every primitive interface and let TCGen extract data as needed. This has two major drawbacks: The significantly increased traffic on the Test Interface demands high data rates and logfiles become very large and, therefore, difficult to handle. But recording as much as possible is quite common, e.g., during field testing.

4.4 TCGen – Test Case Generator

The author assigned the general name TCGen to his test case generator because the basic idea was to create a tool which takes various kinds of input and produces test cases in a configurable language. The currently available beta-version processes logged communication data and generated structure information. It exclusively creates test cases for the TAP application.

TCGen can be separated into two conceptual parts: one interacting with the framework and PCO, the other executing the generation algorithm.

4.4.1 Interaction with Framework and PCO

TCGen is designed as a PCO-Viewer. This means it cannot be used without the PCO-Server. After connecting to the latter the generator selects a filter to receive primitives only and initiates a request for recorded data from a logfile which has to be specified by the user. Now the server transmits all formerly duplicated primitives to TCGen in the same order as they were sent inside the running protocol stack. By accessing the information stored in a matching CCD-Database the data blocks are transformed into structures, and the Entity Graph is used to create suitable routing commands to separate the EUT.

In Figure 15 the explained data flow is represented:

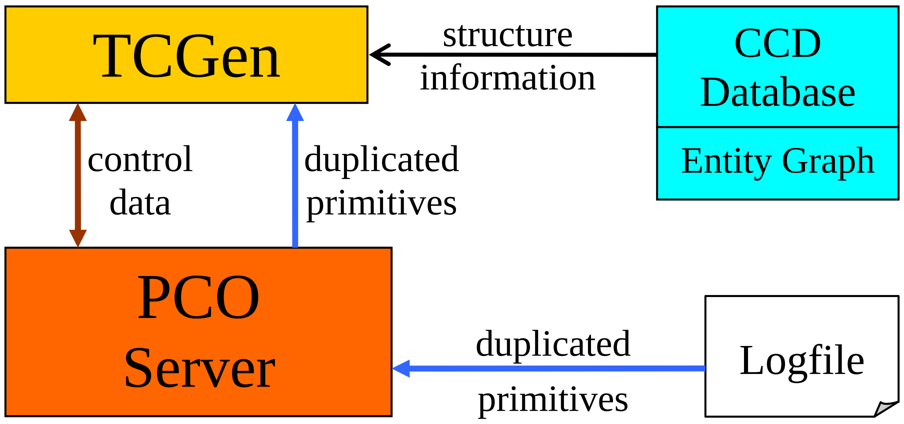


Figure 15 – TCGen Interaction with Framework and PCO

4.4.2 Generation Algorithm

The principle of test case generation out of test session logfiles is quite simple, compared to its actual implementation (see Subchapter 5.5.2). As test oracle the "status quo" at the time of the preceding field test is used. That means the input/output specifications regarding ANSI/IEEE Standard 829 (see Figure 3 on page 17) are created by generating 'send'-commands for all primitives who were sent to the entity-block specified by the user, and by generating 'await'-commands for all primitives emitted by this block, which represents the entities under test. The sequence of these commands corresponds to the primitive order in the logfile, and appropriate test script commands are inserted to simulate longer time differences. To generate the hierarchical content of the primitives (see Table 3, page 20) substructures are defined level by level, with basic type elements set to the actual values from logfile.

Figure 16 demonstrates the algorithm in a pseudo programming language:

```
set level=0;
do
{
     request_logfile();
     set new_elem_found=false;
     while (get_next_primitive(prim))
     {
          if (level==0)
          {
               write_time_differenz_to_last_prim();
               apply_rules(prim);
               write_send_await_command(prim);
          }
          while (get_next_substruct(prim,level-1,substruct))
          {
               write_substract_begin(substruct);
               while (get_next_elem(prim,substruct,elem))
               {
                    set new_elem_found=true;
                    apply_rules(elem);
                    write_elem(elem);
               }
               write_substract_end(substruct);
          }
     }
     set level=level+1;
}
while (new_elem_found);
```

Figure 16 – Central Algorithm of TCGen

Starting with `level=0`, which represents the primitive level, logfile data are periodically re-requested from PCO-Server and the substructures contained in the respective lower level are written to the test case. Called with `-1` as second parameter the function `get_next_substruct()` returns the primitive itself. Function `get_next_elem()` provides access to the direct elements of the given substructure. Depending on the element type `write_elem()` creates an basic type assignment (e.g., `struct->elem1=0x23`) or uses a substructure variable newly declared (e.g., `struct->elem2=new_struct1`), which will be defined during handling of the next level. The algorithm terminates if no more elements were found on the current level, which will always happen after a certain amount of passes since the logged primitives have finite content. In function `write_send_await_command()` applied during the first pass the decision about generating input to be sent versus output to be requested is done by taking the sender/receiver information into account, as provided by the PCO-Server.

There exists one main reason which led to the decision to use the iterative approach instead of recursively creating each primitive structure in one step: Creating the test case sequentially allows instant writing to an actual file system; in other words only minimal context information needs to be managed and kept in memory. In general, the memory consumption of TCGen is very low and independent of the logfile size, since the latter is not handled as a whole. Only the primitive currently received via the Viewer-Interface is examined at a time. The drawback: primitive data are retransmitted many times. This could have performance impacts if the communication between PCO components is realized by, e.g., a network with low data rates, but can be disregarded for a shared memory interface. However, despite the general approach actual implementations of, e.g., the function `get_next_substruct()` will somehow contain recursive parts to efficiently navigate through the primitive structure.

All functionalities previously described allow a user of TCGen to create test cases which will instruct the execution tool (TAP) to send and expect primitives containing exactly the element values as recorded by PCO. A run with such test scripts will most likely fail, e.g., because of temporary values created randomly by mobile or base station, or because of elements which are irrelevant in the given context but were logged with values now awaited. Also the names of some primitives differ from the ones used for the same task during simulation testing on a PC. In brief, certain modifications to the original data have to be applied during the generation process. In the pseudo code on page 40 the function `apply_rules()` implements such a mechanism based on a description of rules which has to be provided by the user. Figure 17 contains the formal specification for a set of rules understandable by TCGen. The key words printed in italics are defined in the BNF of the test script language (see, e.g., Figure 7 on page 28).

```
<RulesDescription> ::= <Options> <RuleList>
<RulesList>        ::= <Rule> [<RulesList>]
<Rule>             ::= <SkipRule> | <ChangeRule>
<Options>          ::= 'options' ['max_timegap='<Integer>] ['timeouts='<Integer>]
<SkipRule>         ::= 'skip' 'primitive='<Mask> ['param='<Mask>]
<ChangeRule>       ::= 'change' 'primitive='<Mask> ['param='<Mask>]
                       <ChangeStatement>
<ChangeStatement>  ::= <Test-Script-Statement
                       [including one ore more <Placeholder>]>
<Mask>             ::= <String>['*']
<Placeholder>      ::= '%v'
```

Figure 17 – Partial BNF of TCGen Rules Descriptions

Beside settings for several options such a description contains various ‘skip’- and ‘change’-rules, applicable to a whole primitive or a specific element. By using wildcards equal rules can be combined. Skipped primitives will not be mentioned in the test case; values of skipped elements will be ignored. Concerning the specification of changes which shall be applied all possibilities offered by the test script language can be used, since the test case generator will just substitute the original value by the specified statement. If placeholders are found the value from the logfile is inserted for each ‘%v’. This method is very easy to implement and very powerful at the same time. See Appendix A.1 for examples.

A complete test case as expected by the TAP does not only contain the input/output specifications. Moreover, initial FRAME routing commands have to be generated in a way that the communication to and from the entity-block under test is redirected to TAP. If Entity Graph information is available for the specific protocol stack, an algorithm similar to the one described in Subchapter 4.3.2 is used to create the appropriate commands. Unfortunately, for some older protocol stack versions, which still have to be supported, no such graph exists. TCGen solves this problem by performing an additional pass during which only the sender/receiver information of the logged primitives is considered to temporarily build up a communication graph. This, of course, may lack certain primitive interfaces which were not used or not duplicated during the field test.

4.5 TAP – Test Application Process

As already explained, TAP is primarily used to run test cases on a PS simulation. But because of the generic framework the behavior of the entity/ies under test can be as well checked using an actual mobile phone. In any case the routing

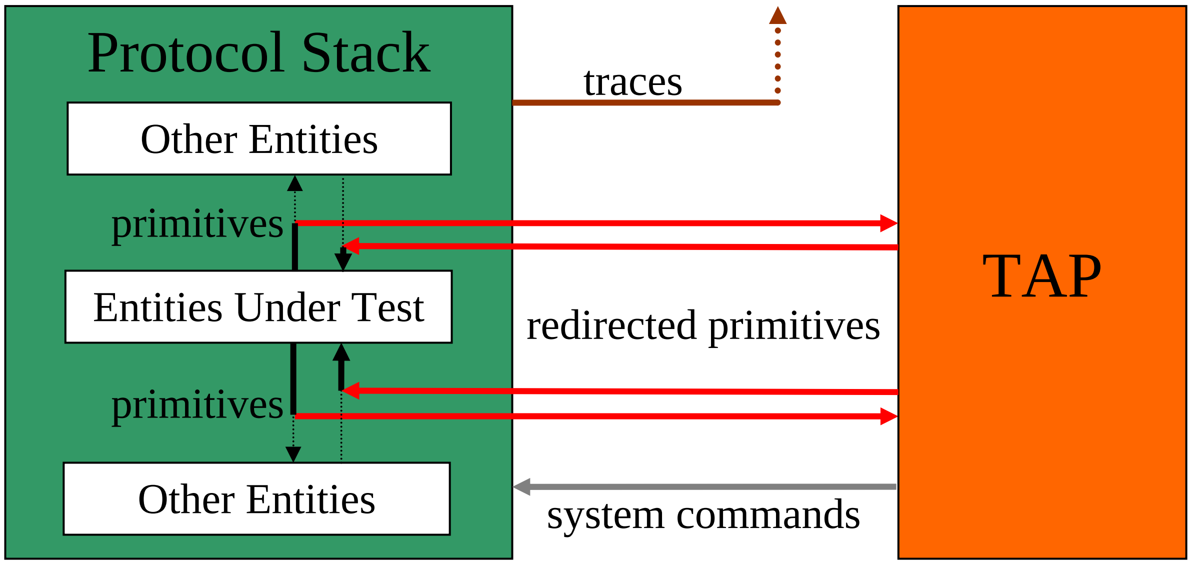


Figure 18 – TAP Communication with Separated EUT

commands specified in the test case are sent to FRAME on protocol stack side, which leads to the situation presented in Figure 18. The EUT are isolated in such a way that all primitives sent from one of them to another outside the block are redirected to the TAP, and those that are sent from other entities are completely discarded.

After this initialization the TAP can send data to the tested entities and evaluate answer-primitives regarding the specifications in the test case. After the test run a short protocol is created, stating whether the case passed or failed. More information about problems which occurred during the test can be found out by, again, examining the produced traces. These are not handled by the TAP, but since usually PCO is started in parallel, trace and duplicated primitive data are visualized and logged as already described in Subchapter 4.3.

5 Practical Realization of the Test Case Generator

In this chapter we describe the actual implementation of parts of the software theoretically examined in chapter 4.

Beside general conditions like the used environment and coding rules the diverse software layers and their functionalities will be explained. Furthermore we will have a look at specific implementational details of the PCO tool and the test case generator developed together with this diploma thesis.

5.1 Technical Preconditions

While the generic framework as an operating system abstraction provides ports for several OS kinds (see Subchapter 5.3), all test tools are currently developed for the MS-Windows operating system only due to the equipment used by the testers. However much effort has been put into keeping most of the source code portable, as explained more deeply in Subchapters 5.4.1 and 5.5.1.

The concrete hardware platform used while developing PCO and TCGen consisted of an INTEL PC (see [INTEL]) running WINDOWS NT 4.0.

C++ was used as programming language together with several generally available software libraries like the MICROSOFT FOUNDATION CLASSES (*MFC*, see [MFC]), the *Standard Templates Library* (*STL*, see [STL]) and a module for parsing *XML*-files. Compiler and linker have been taken from the software bundle MICROSOFT DEVELOPER STUDIO 6.0.

Furthermore, the software PC-LINT from GIMPEL SOFTWARE (see [GIMPEL]), a static analysis tool focusing on the C and C++ languages, has been integrated in the build process.

5.2 Coding Rules

The TEXAS INSTRUMENTS BERLIN AG provides several global rules concerning software programming. The following notes represent some of the more important regulations:

- Writing ANSI-C/C++ style is highly requested.
- Several templates for preceding comments in files or before functions have been provided and have to be used.
- Include files should be functionally organized, i.e., declarations for separate subsystems should be in separate files.
- Accidental double-inclusion of include files has to be avoided, e.g., by dedicated defines.
- Warnings that occur during compilation should be minimized.
- *GNU* (see [GNU]) compatible makefiles should be provided for each project with which it must be possible to compile each part of the code as well as to build the whole system.
- Naming:
    - All names (of, e.g., variables or functions) should be meaningful and give an idea of their application.
    - Function names must consist of the component name, an underscore, and any other sequence of lower case letters, digits, and underscores.
    - The names of new data types defined with the 'typedef' command have to start with a "T_".
    - Variable names must consist of any sequence of lower case letters, digits, and underscores.
    - Constant names must consist of upper-case letters that may be sepa-

rated by “_” characters and must not begin with “T_”.

- Coding:
  - Avoid labels and the corresponding “goto” statement.
  - In new blocks the opening and closing parenthesis have to be put in the same column as the code before the block. The code within blocks has an indent of two white spaces. Comment lines have to be indented as if they were code lines.
  - In switch constructs the statements after a label again are shifted two characters to the right. If the “break” is omitted though there is at least one statement after the label, a comment must indicate this as intentional.
  - As source code debugging of macros is difficult macros should only contain a few statements.

Due to the fact that these general guidelines were worked out for development using the non-*object-oriented* language C some “internal” rules have been additionally defined and applied by the author:

- Class names start with a capital letter followed by any sequence of lower or upper case letters, digits, and underscores.
- Static module variables and member variables of classes are prefixed by “m_”.
- Call by reference should be the preferred method when passing objects as function parameters.
- The key word `const` shall be applied to variables and member functions in any possible case.

5.3 Frameworks and Software Layers

This subchapter describes the implementation of frameworks used by the test tools. The description will be quite general since these software parts were not implemented by the author.

FRAME: As already mentioned in Chapter 4.1 FRAME is an abstraction of the operating system actually used. In the case of embedded mobile devices this will be a *real time operating system* (RTOS) like NUCLEUS or VXWORKS. But also *desktop operating systems* like MS WINDOWS are supported. This is realized by a part of the FRAME called *OS Layer* which has to be implemented specifically for each operating system, but finally provides a common *Operating System Interface* with functions like `os_CreateTask()` to create a new OS *process*. All further layers (as shown in Figure 19) are implemented only once in a generic fashion, since they need not to know which operating system is used but can just call into the OS Layer.

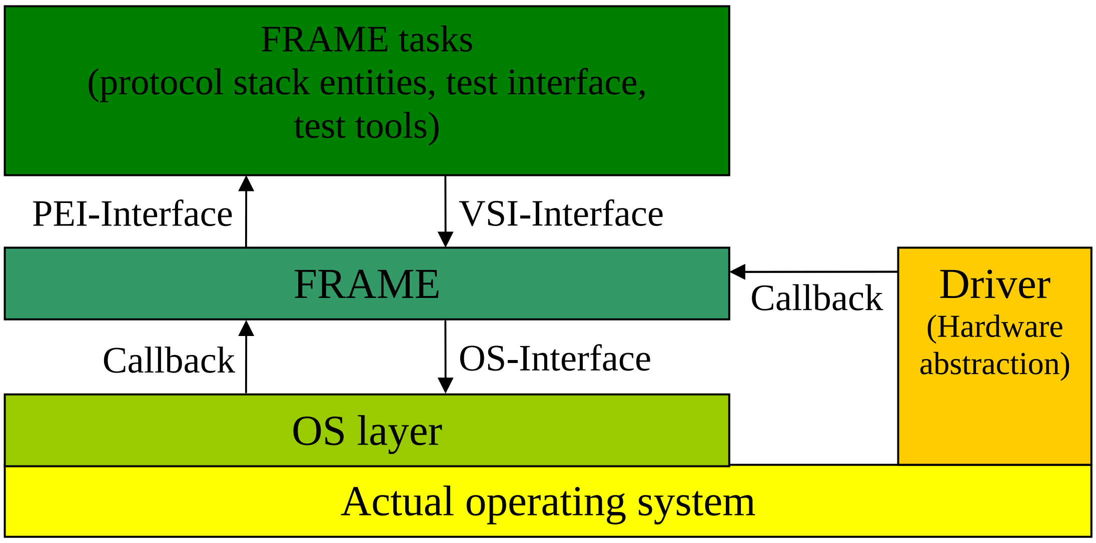


Figure 19 – FRAME Architecture

While so far “only” an OS abstraction has been achieved, FRAME offers several extensions – functionalities which cannot be found in all operating systems like *memory supervision*, *passive task model*, timer configuration and last but not least routing and tracing.

The common way to use FRAME is implementing one or more FRAME tasks based on a template source file containing empty bodies of all needed *PEI*-functions. The *Protocol Stack Entity Interface* (PEI) consists of a set of callback functions like `pei_init()` or `pei_primitive()` known to FRAME, which will be called whenever an appropriate event occurs – e.g., during initialization or if a primitive has arrived. As the name implies, this interface has been originally designed exclusively for protocol stack entities; but for testing purposes it makes sense to reuse it for the Test Interface and test tools like TAP. The *Virtual System Interface* (VSI) on the other hand offers several FRAME functions to the tasks, which might be handled inside the FRAME or forwarded via the OS-Interface to the actual operating system. Examples are `vsi_c_send()` or `vsi_m_new()` to send a primitive to another task or to allocate memory, respectively.

The whole FRAME functionality is collected in several libraries which have to be linked to the task sources to obtain a protocol stack. The Test Interface responsible for outbound communication is implemented as a FRAME task as well. For WINDOWS OS, FRAME consists of three *Dynamic Linked Libraries* (DLLs) including shared variables to handle one or more “connected” test tools. Furthermore it was necessary to wrap the Test Interface into a standalone executable to keep this task running independently from the start/exit of tools. It takes parameters specifying the communication type.

FRAME supports various system commands (see examples in Table 4) which can be sent via VSI functions, e.g., for dynamic re-configuration of communication paths or to enable/disable tracing for specific entities. In return system messages like `”SYSTEM WARNING: Receiver process entity un-`

`known"` are emitted. Details may be looked up in [FRAME].

Command	**Function**	**Example (→ .. sent to)**
`RESET`	Reset the receiver entity	`RESET → GMM`
`STATUS <resource>`	Request information from the receiving entity	`STATUS QUEUE → RR`
`DUPLICATE <org destination> <new destination>`	Duplicate primitives send from receiver to <org destination> to <new destination>	`DUPLICATE MM PCO → RR`
`REDIRECT <org destination> <new destination>`	Route primitives send from receiver to <org destination> to <new destination> instead	`REDIRECT GMM TAP → GRR`
`TRACECLASS <class-mask>`	Configure receiver entity to trace only information matching the <class-mask>	`TRACECLASS FFFF → RR`

Table 4 – Examples of FRAME System Commands

(V)CMS: The *(Virtual) Condat Multitasking System* takes an important role in the FRAME implementation on a desktop operating system. It functions as an additional layer between the OS Layer and the operating system (see Figure 20) and provides the actual adaptation. For example the OS Layer function `os_CreateTask()` calls the VCMS function `p_create()` which will finally use a system call of MS WINDOWS or Linux – the two systems currently supported – to create a process. Obviously only one implementation of the OS Layer is needed for desktop operating systems.

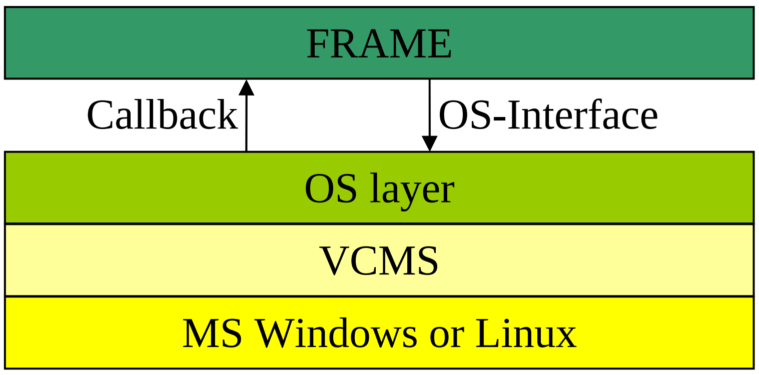


Figure 20 – FRAME Using VCMS

This special architecture has evolved historically. The original CMS is an actual operating system running on gateways connected to multiple mobile devices for testing purposes. Implementations exist also for, e.g., MOTOROLA M68332 and the INTEL architecture. One main feature of CMS is the support of communication queues through which CMS processes can exchange data.

To offer the comprehensive CMS interface and capabilities also to applications designed for MS WINDOWS (because the CONDAT AG testers primarily worked on WINDOWS PCs) a virtual version called VCMS had been developed – not accessing the *processor* directly anymore. Later on, an adaptation for Linux followed. VCMS is available as dynamically loadable library using shared memory to implement the data queues.

During the FRAME design phase, it was decided to use VCMS due to the fact that many of the features required were already implemented by the Virtual Condat Multitasking System – including the OS adaptation. See Source Code 1 on page 84 for details concerning the CMS interface.

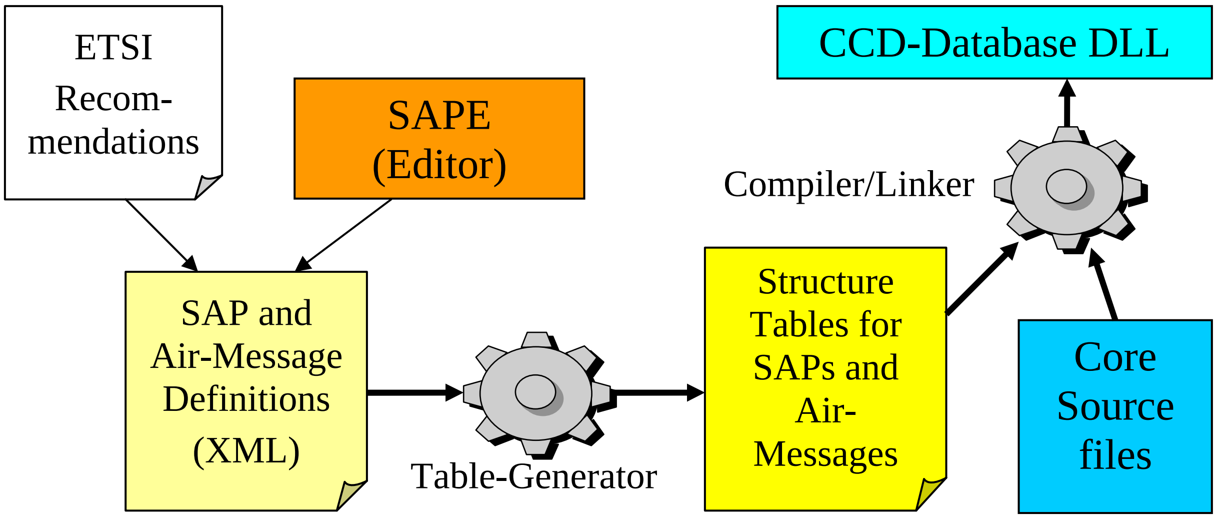


Figure 21 – Creation of the CCD-Database

CCD-Database: The database containing all information about primitive and air message structures as described in 4.1 is realized as a WINDOWS-DLL (per default: `ccddata_dll.dll`). Figure 21 explains the creation process of such a library: Supported by a dedicated editor (see [SAPE]) developers create/edit definition files for Service Access Points and Air-Messages regarding the ETSI recommendations. The XML-files are transformed into C-tables by a special generator tool, compiled and finally linked together with core source files. In the latter the functional interface to access particular data is defined. It contains, for example, the functions `cde_prim_first()` and `cde_prim_next()` to read out primitive information. Source Code 2 on page 85 gives an impression of the functions available.

Each time something is changed in one of the XML-files, a new DLL will be created automatically during the protocol stack build. So, for correct results testers should always receive a matching library together with a certain PS image-file. Concepts exist to include the DLL in the image-file.

Entity Graph: For the representation of existing communication interfaces between entities in a protocol stack the second and simpler graph of the two described in Subchapter 4.2 was chosen. In particular, a two-dimensional data field corresponding to the so-called adjacency matrix of the undirected graph is maintained as part of the already described CCD-Database. This is an $n \times n$-matrix with n being the number of nodes (existing entities). On position (i,j) it contains the number of edges between node i and node j – 0 if no communication interface exists between entity i and entity j, 1 otherwise. See Source Code 3 on page 86 for an actual example. As can be seen there, the adjacency matrix is symmetric for undirected graphs. Source Code 4 (page 86) contains the small but sufficient set of functions to access the Entity Graph.

The adjacency matrix is currently maintained manually, and no concept exists so far to generate it, e.g., out of MSC charts. On the other hand, the knowledge about communication partners of entities is present inside the framework on protocol stack side right after the initialization phase. During the latter all entities open their communication channels. So, tools like PCO or TCGen could request the Entity Graph from FRAME at runtime instead of using precompiled CCD-Database information. However, information about existing SAP-s and their names will, as well, not be available this way, and some efforts are still needed to include this feature in future versions.

The following picture (Figure 22) summarizes all previous explanations in a general overview about the software layers and usage relations:

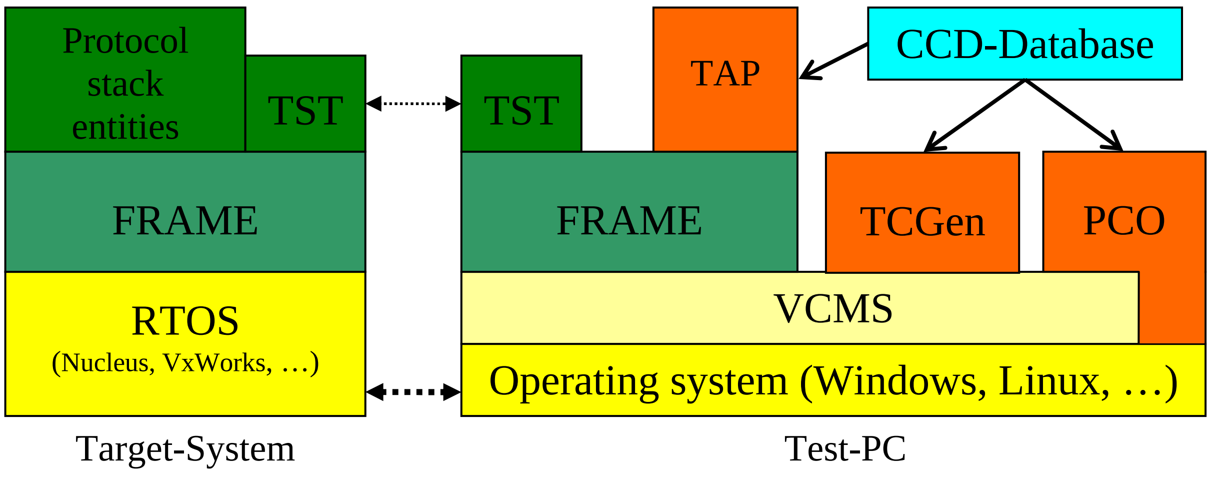


Figure 22 – Software Layers

It can be seen how the test tools TAP, PCO and TCGen utilize the frameworks – TAP as a FRAME task, PCO and TCGen as direct clients of VCMS. All tools access information about the PS running on the connected target system by loading the CCD-Database-DLL and calling appropriate functions. The usage relationship of the tools developed by the author will be explained in more detail in the next subchapters – including a description of the PCO shortcut to the operating system.

Finally it shall be stated that the whole frameworks software has been implemented using the original C programming language, unlike, e.g., the test case generator. This is mainly because stable compilers for all used real time operating systems exist for this language, but C++ will probably be introduced in the near future.

5.4 PCO – Point of Control and Observation

PCO is the testing tool capable of visualizing traces sent out of the protocol stack under test and, therefore, is used during various test scenarios from field tests to TAP executions (see Subchapter 3.2). Not every part of PCO is needed each time, but a common ini-file is always used to store general and component specific settings. See Source Code 6 (page 87) for a default version.

Although it would have been a possibility PCO was not implemented as a FRAME entity. This design decision has originally been made with the intension to create a universal tracing tool. In fact the core implementation does not depend on any specifics of the test frameworks described in Subchapter 4.1. But till now only one final PCO application exists, which needs specific knowledge of the FRAME – since it is used to visualize information about FRAME entities inside a protocol stack.

The different PCO components – server, controller and viewers – communicate via the VCMS interface (named CMS queues, see Figure 25 on page 56) since all of them are implemented running on top of the VCMS layer as can be seen in Figure 23. The shortcut to the actual operating system also mentioned there is drawn due to the fact that the GUIs of PCO are currently not implemented using portable libraries, but the WINDOWS specific MFC.

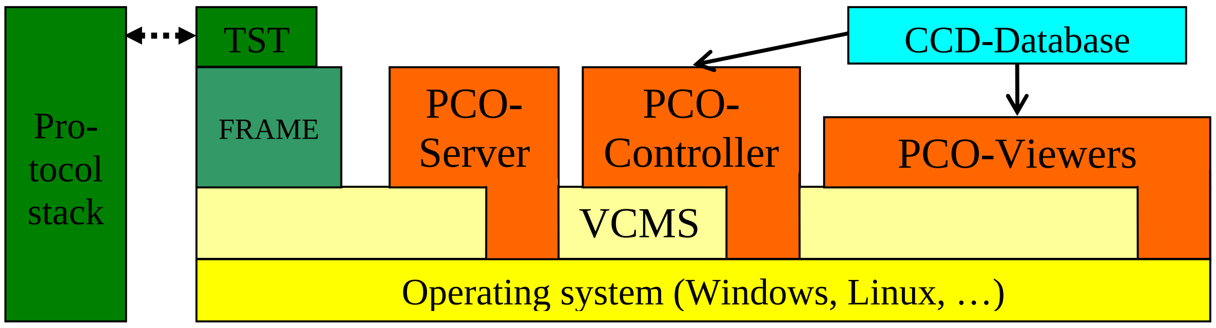


Figure 23 – Software Layers Used by PCO

The PCO-Server is the only part which connects to FRAME on tool side to send/receive primitives. The CCD-Database is used by viewers and controller.

5.4.1 *Object-Oriented Approach*

In contrast to the described parts of the framework, PCO has been implemented using the language C++ which supports object-oriented design. This feature is mainly used to separate the functionalities and allows for a generic core with increasingly specialized inheritors.

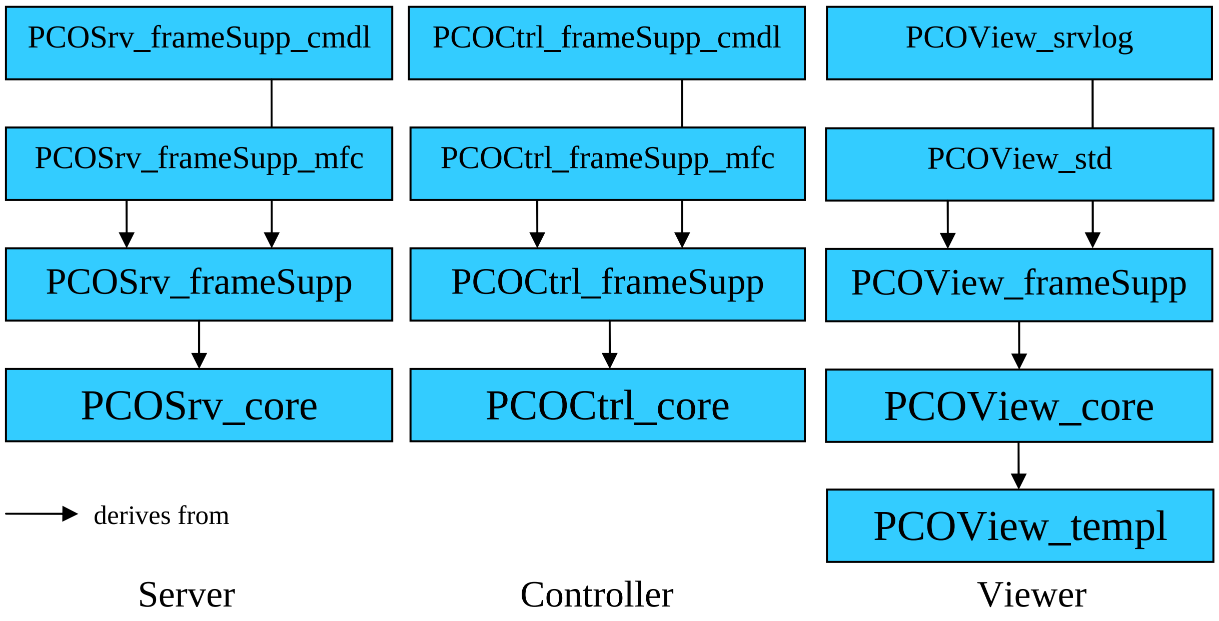


Figure 24 – PCO Class Hierarchies

In the overview shown in Figure 24, it can be seen that for each PCO part a class suffixed by `_core` exists. These contain general PCO functions, e.g., for creating communication queues or exchanging data with a common header (see Source Code 7, page 87).

The core classes are independent of the operating system by using VCMS and ANSI-C++ and do not contain any information concerning the FRAME test concepts. This specific knowledge is introduced in the derived classes with names ending on `_frameSupp`. These are still OS independent, but the separation maintains the option of PCO to be adapted to other needs without to many changes. All classes derived from the FRAME-dependent layer are final classes adding mainly GUI specific functionalities.

5.4.2 Components and Communication

PCO-Server: The server as the fundamental part of PCO receives traces and duplicated primitives from the Test Interface entity running on top of the FRAME and it can, in turn, send system commands. For this purpose a special FRAME extension called *TST reception interface* provided as a library is used offering functions for sending data and to register a CMS queue for receiving (see Source Code 5, page 86). The receiver queue will always be named “PCO”. Source Code 8 on page 89 contains the actual implementation of the registration.

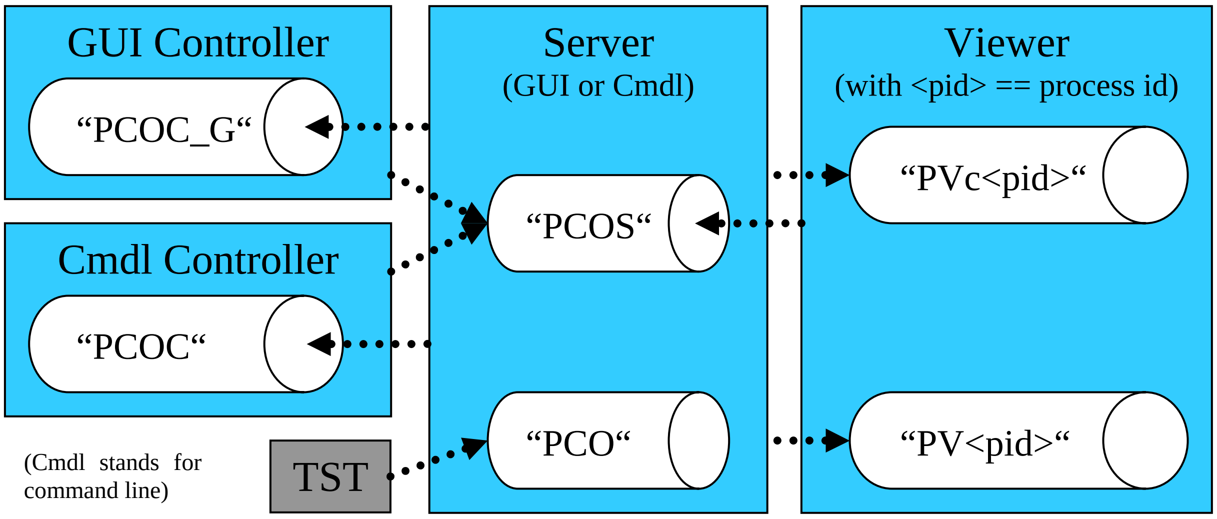


Figure 25 – CMS Queues Used for Inter-PCO-Communication

For internal control messages coming from the controller or any kind of viewer, another queue (“PCOS”) is used to be always able to react (see Figure 25). The controller will usually request a change of the server state (`PCO_RUNNING`, `PCO_STOPPED` or `PCO_LOGFILE`) or submit FRAME system primitives to be forwarded to the protocol stack. A viewer has, first of all, to connect to the server resulting in its queue names and other information being added to the client list. Afterwards it can, e.g., ask for data from a logged session or subscribe for live data emitted by a running PS.

Internal communication follows the general scheme of sending control data and waiting for a response; this can be either `PCO_OK` or `PCO_ERROR`, the latter with an error code. Table 5 contains an overview about selected PCO control messages. More details can be looked up in [TRACING].

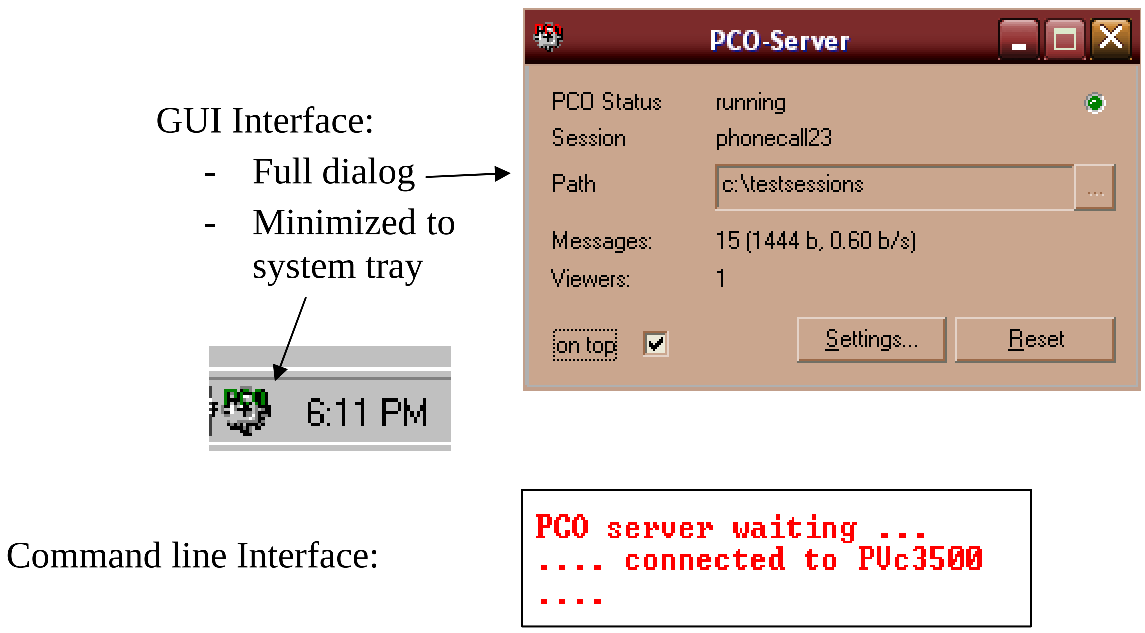
Message	Parameters	Sent from … to …
`PCO_CONNECT`	<CMS queue for receiving traces/primitives>	`Viewer → Server`
`PCO_SET_FILTER`	<names of entities whose data shall not be forwarded> [<id to select traces, primitives or both>]	`Viewer → Server`
`PCO_START_TESTSESSION`	<name of test session>	`Controller → Server`
`PCO_ERROR`	<id of message which caused the error> <error code>	`ANY → ANY`
`PCO_OPEN_LOGFILE`	<name of test session> [<start entry nr> <end entry nr>]	`Viewer → Server`
`PCO_LOGFILE_COMPLETE`	-	`Server → Viewer`

`PCO_SYNCHRONIZE`	<time stamp>	`Viewer → Server`
		`Server → Viewer`

Table 5 – Selected Control Messages of the PCO-Protocol

In its initial state (`PCO_STOPPED`) the server will discard all received primitives until either viewers have subscribed or the state changes to `PCO_RUNNING`. In the latter case, the logging process into a previously specified file will be started. The state `PCO_LOGFILE` is used to load a logfile into the server which then can be replayed through all connected viewers in parallel. Forwarding data to viewers is – for performance reasons – the only PCO communication without acknowledgement. The concrete format of PCO logfiles is explained in Subchapter 5.4.4.

Currently two complete implementations of the server exist – a command line version (`PCOSrv_frameSupp_cmdl`, `pcod.exe`) and a GUI server for MS WINDOWS (`PCOSrv_frameSupp_mfc`, `pco_srv.exe`). Figure 26 shows the appearances of these variants. While the command line executable prints out strings to reflect the current state, the GUI variant changes its colors and can show more information in a dialog window. Except the GUI interface both servers support the same functionalities. The command line version was mainly implemented to ease an eventually change to Linux as operating system. Only one server can run at any given time.



PCO Status running
Session phonecall23
Path c:\testsessions
Messages: 15 (1444 b, 0.60 b/s)
Viewers: 1

```
PCO server waiting ...
.... connected to PVc3500
....
```

Figure 26 – PCO-Server Appearances

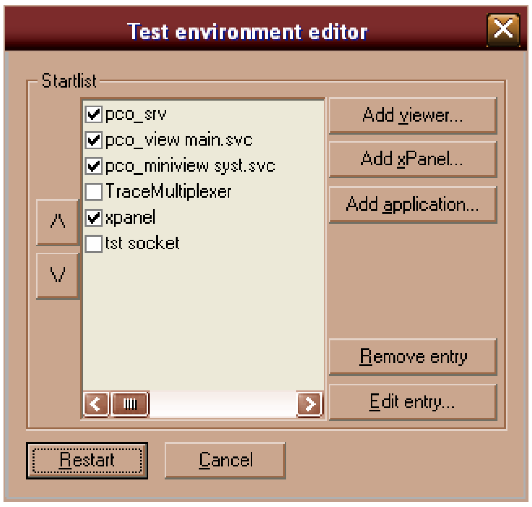
PCO-Controller: Like the server, the controller exists in a GUI (`PCOCtrl_frameSupp_mfc`, `pco_ctrl.exe`) and a command line (`PCOCtrl_frameSupp_cmdl`, `pcoc.exe`) version, the first being MS WINDOWS specific. Figure 30 on page 62 gives an impression of the graphical user interface. The common functionality of both versions implemented in the base classes `PCOCtrl_core` and `PCOCtrl_frameSupp` comprises the manipulation of the server state and the forwarding of system primitives by sending corresponding messages to the "PCOS"-queue. But controllers receive replies in queues named differently: "PCOC_G" for the GUI and "PCOC" for the command line variant (see Figure 25, page 56). This permits the usage of `pcoc.exe` even when the GUI controller is running, which makes sense because the command line executable is heavily used in scripts to automate several testing tasks. Figure 27 shows the `pcoc`-parameters representing the available functions.

```
pcoc {[-n <server-queue-name>]
        {start <session-name> | stop | open <session-name> |
         send <receiver> <system-primitive> | close | exit |
         msend <ASCII-file with 'receiver system-primitive'-lines>
         spath <session-path> | distrib}
      | -ver}
```

Figure 27 – PCO Command Line Controller Parameters

The PCO controllers offer the user several options to specify system commands, which will be sent to the connected PS. At the command line he has to input them manually; this is also possible within the GUI. However, the graphical interface offers also a list with predefined commands which can be combined freely and sent by pressing one button. See 5.4.4 for further possibilities.

All these functions were not integrated into a viewer as there can be more than one running at a time while controlling should be a central task. Moreover, the GUI version is also used to manage the start and exit of a set of applications, including the other PCO components and, for example, the xPanel. This test environment can be changed by the user in a dedicated dialog (see Figure 28).

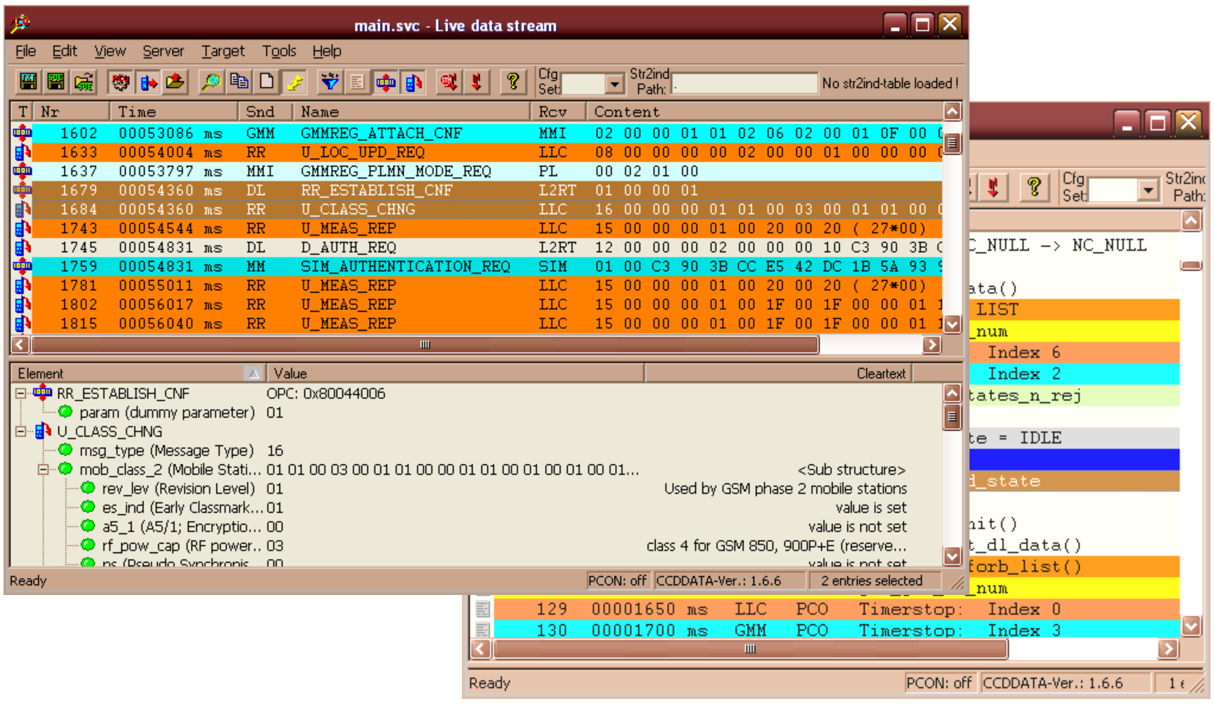


Startlist
- [✓] pco_srv
- [✓] pco_view main.svc
- [✓] pco_miniview syst.svc
- [ ] TraceMultiplexer
- [✓] xpanel
- [ ] tst socket

Figure 28 – Test Environment Dialog of PCO-Controller

Finally the GUI controller is the application where the type of communication (serial connection, shared memory, etc.) to be used to access the protocol stack can be selected – including all parameters. The choices are used as parameters to a call of the Test Interface executable.

PCO-Viewers: Any application deriving from one of the viewer classes as explained in the next subchapter is considered as a PCO-Viewer. The first ever implemented was the so-called *Standard Viewer*. It is an MFC based WINDOWS executable with an interface as shown in Figure 29. This viewer represents traces and primitives as entries in a list view and uses the CCD-Database to interpret/decode their contents, which then is visualized in a tree view below the entry list. Users can specify several filter conditions (e.g., by the primitive *operating code* (OPC)) and select different colors for each sender entity. This is very important given the fact that during heavy testing easily more than 10 000 traces per minute can be produced. All entries can be copied to the *clipboard* or exported into several file formats.



```
T  Nr    Time          Snd  Name                        Rcv   Content
   1602  00053086 ms   GMM  GMMREG_ATTACH_CNF           MMI   02 00 00 01 01 02 06 02 00 01 0F 00
   1633  00054004 ms   RR   U_LOC_UPD_REQ               LLC   08 00 00 00 00 02 00 00 01 00 00 00
   1637  00053797 ms   MMI  GMMREG_PLMN_MODE_REQ        PL    00 02 01 00
   1679  00054360 ms   DL   RR_ESTABLISH_CNF            L2RT  01 00 00 01
   1684  00054360 ms   RR   U_CLASS_CHNG                LLC   16 00 00 00 01 01 00 03 00 01 01 00
   1743  00054544 ms   RR   U_MEAS_REP                  LLC   15 00 00 00 01 00 20 00 20 ( 27*00)
   1745  00054831 ms   DL   D_AUTH_REQ                  L2RT  12 00 00 00 02 00 00 00 10 C3 90 3B
   1759  00054831 ms   MM   SIM_AUTHENTICATION_REQ      SIM   01 00 C3 90 3B CC E5 42 DC 1B 5A 93
   1781  00055011 ms   RR   U_MEAS_REP                  LLC   15 00 00 00 01 00 20 00 20 ( 27*00)
   1802  00056017 ms   RR   U_MEAS_REP                  LLC   15 00 00 00 01 00 1F 00 1F 00 00 01
   1815  00056040 ms   RR   U_MEAS_REP                  LLC   15 00 00 00 01 00 1F 00 1F 00 00 01

Element                          Value                                                    Cleartext
RR_ESTABLISH_CNF                 OPC: 0x80044006
   param (dummy parameter)       01
U_CLASS_CHNG
   msg_type (Message Type)       16
   mob_class_2 (Mobile Stati...  01 01 00 03 00 01 01 00 00 01 01 00 01 00 01 00 01...    <Sub structure>
      rev_lev (Revision Level)   01                                  Used by GSM phase 2 mobile stations
      es_ind (Early Classmark... 01                                                     value is set
      a5_1 (A5/1; Encryptio...   00                                                 value is not set
      rf_pow_cap (RF power..     03                          class 4 for GSM 850, 900P+E (reserve...

Ready                                  PCON: off | CCDDATA-Ver.: 1.6.6 | 2 entries selected
```

```
C_NULL -> NC_NULL

ata()
 LIST
_num
  Index 6
  Index 2
tates_n_rej

te = IDLE

d_state

nit()
t_dl_data()
forb_list()
_num
   129  00001650 ms   LLC   PCO   Timerstop:   Index 0
   130  00001700 ms   GMM   PCO   Timerstop:   Index 3

Ready                                  PCON: off | CCDDATA-Ver.: 1.6.6 | 1
```

Figure 29 – The Standard PCO Viewer

After PCO was widely used, further viewers were developed, partly by external companies – one, for example, monitoring only specific parameters of the network, and another controlling Robustness Tests by evaluating traces. A command line dumper has been added by the author and many others, e.g., a *HTML*-viewer are imaginable. Moreover, the capabilities of the Universal Viewer Approach are used inside the PCO-Server to implement the logging mechanism: A hidden internal viewer (implemented by `PCOView_srvlog`) derived from the FRAME dependent viewer layer (see Figure 24, page 55) is used to support several historical logfile types (containing, e.g., pre-interpreted traces) beside the standard PCO format. To enable automatic reactions of PCO-Controller upon restart of the protocol stack, this component was also enhanced by an internal viewer to receive FRAME system messages.

5.4.3 The Viewer-Interface

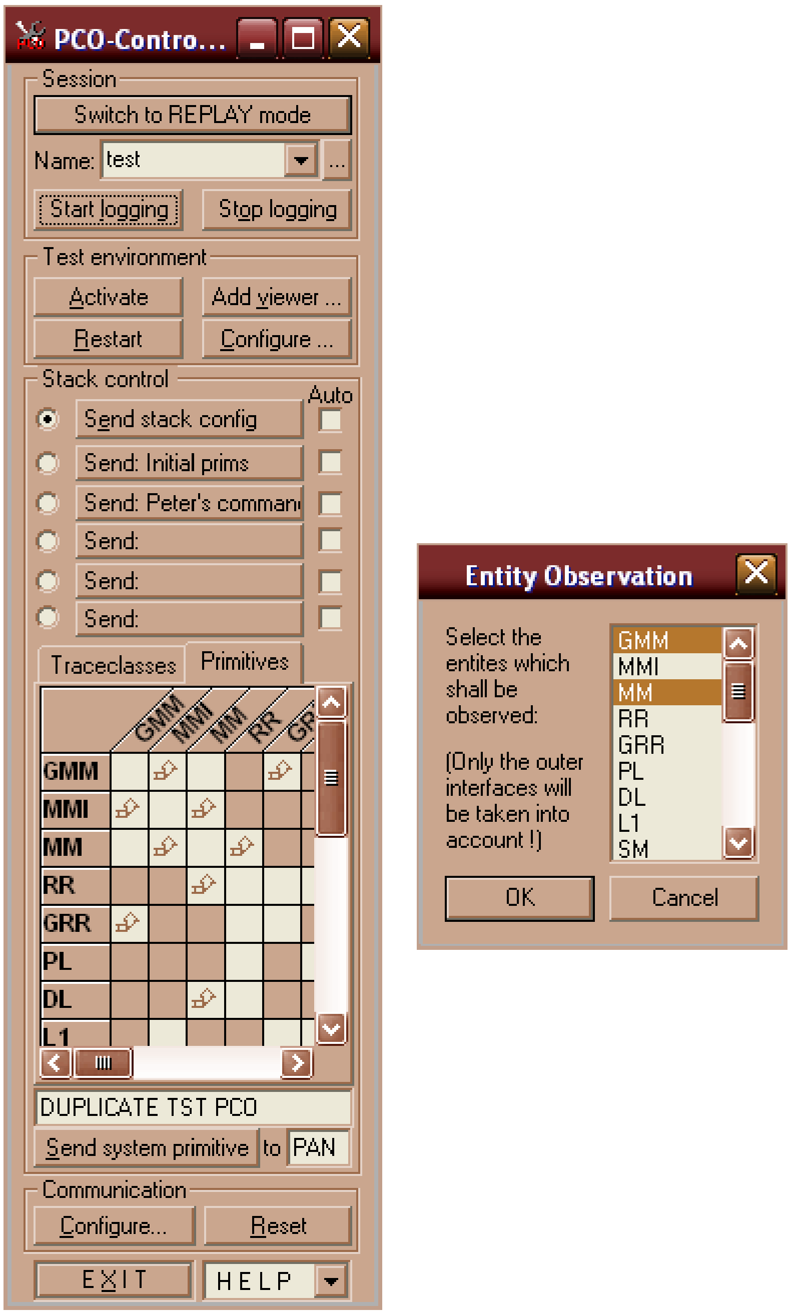
To create a PCO-Viewer a new application has to comply with the minimal requirements provided by one of the base classes for viewers (see Source Code 9, page 89): `PCOView_templ`, `PCOView_core` or `PCOView_frameSupp`.

While deriving from the first of these classes leaves even the creation of CMS queues to the final inheritor, `PCOView_frameSupp` contains all necessary functionalities to interact inside a FRAME based test environment. Central part of the methodology are the virtual functions `dispatch_message()`, `interprete_message()` and `on_data()`. The first is for handling control messages. Default treatment of, e.g., PCO_EXIT is provided in `PCOView_core`, which can be overwritten by final viewer classes. The second, `interprete_message()`, is called whenever new traces/primitives arrive to extract, e.g., sender, receiver and data part who will afterwards be passed to `on_data()`. While at least `PCOView_frameSupp` includes an implementation of this interpretation concerning the FRAME format, the third function has always to be implemented by the viewer application – matching the actual needs.

With `pco_view.lib` a library containing all core sources is provided, which has to be linked to viewer implementations. The parts using the CCD-Database are compiled into separate object files to enable viewers without CCD support.

5.4.4 *Logging of Entity Communication*

As Figure 23 on page 54 already revealed, PCO-Controller utilizes the CCD-Database, too. In detail, it needs Entity Graph information to provide the user with a matrix as shown in Figure 30, which allows selection of primitive duplication for all communication interfaces available.



Name: test

	GMM	MMI	MM	RR	GR
GMM		✓			✓
MMI	✓		✓		
MM		✓		✓	
RR			✓		
GRR	✓				
PL					
DL			✓		
L1					

DUPLICATE TST PCO

to PAN

Select the entites which shall be observed:

(Only the outer interfaces will be taken into account !)

GMM
MMI
MM
RR
GRR
PL
DL
L1
SM

Figure 30 – GUI of PCO-Controller with Matrix and Observe Dialog

Clicking on it will result in automatic translation of the selections into matching system commands. For example, choosing the cross section between entities MMI and GMM is realized by the string "`DUPLICATE GMM PCO`" sent to the protocol stack. Not existing connections are visualized as disabled and grayed-out fields. A second matrix exists for selecting trace-classes (see Subchapter 4.2) to be enabled/disabled for specific entities.

The Entity Graph is, furthermore, used after the user has selected several entities in a dedicated observation dialog. These entities are considered as a block and, using the algorithm explained in Subchapter 4.3.2, appropriate settings in the matrix are generated to request duplication of all outer interfaces.

The activities described so far result in primitives arriving at PCO-Server and being displayed in connected viewers, depending on the used filter. To actually log these data a session name has to be selected in the controller GUI and the "Start logging" button has to be pressed. Now, the raw binary data coming from protocol stack is written into an actual file. But, mainly in order to ease the later access to it, the logfile format consists of additional information elements: a version identifier, an initial number representing the count of trace/primitive entries contained and continuous jump marks. This way, it is possible to request only dedicated parts without the need to examine the whole file, which significantly increases performance in case of large logfiles.

5.5 TCGen – Test Case Generator

Unlike PCO, the test case generator presented in this document does not use any WINDOWS specific libraries but exclusively utilizes the VCMS layer for process and queue management and ANSI-C++ functions for file handling. It can, therefore, be considered as widely portable; although, only executables for the MICROSOFT operating system (`tcgen.exe`) have been created so far. As described in the theoretical conception TCGen is a construct consisting of two major parts, whose implementation will be examined in the next subchapters.

5.5.1 A Special PCO-Viewer

The object oriented approach of PCO (see Subchapter 5.4.1) has been re-used consequently for TCGen. Since the tool had been designed to take PCO logfiles as input which contain traces and primitives, each formatted with a FRAME header, it made sense to derive the core class of the generator, `TCGenerator_core`, from `PCOView_frameSupp`. The following picture (Figure 31) presents an overview of the whole class hierarchy:

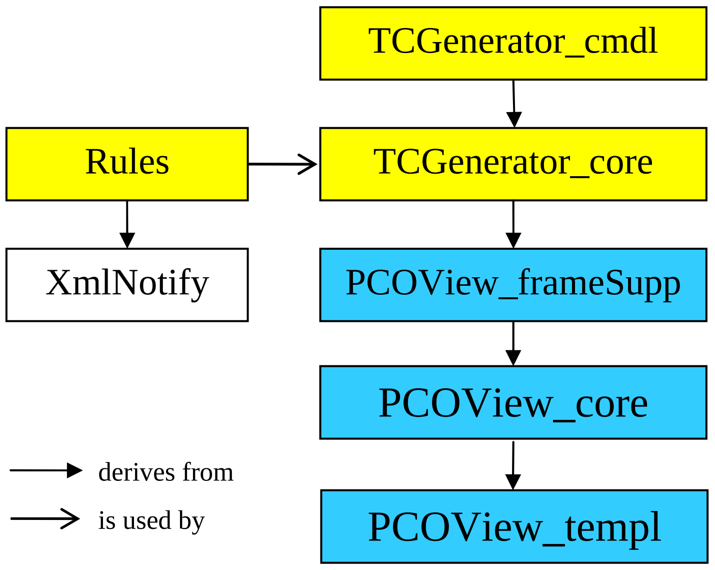


Figure 31 – TCGen Class Hierarchy

By inheriting in the mentioned way, TCGen became a special PCO-Viewer with an already implemented `interpret_message()` function. See Source Code 11 on page 91 for the core class declaration. Only `on_data()` had to be defined individually to fulfill the requirements of the Viewer Interface. This function is the place where the actual generation process takes place, as described in Subchapter 5.5.2. Some more base class methods like `on_connected()` were overwritten to, e.g., add specific behavior in case of successful connection to PCO-Server. The implementation of the user interface has been moved to the child class `TCGenerator_cmdl`. This will simplify the substitution of the current command line interface by a graphical one in future. But additionally, the core class contains auxiliary functions used during generation, configuration methods like `set_rules()` and finally `analyze()` and `generate()`, both taking at least the name of a PCO logfile as parameter to start the actual algorithm – putting out a communication table (see Figure 36, page 71) in analyze-mode or generating a TDC test case in generate-mode.

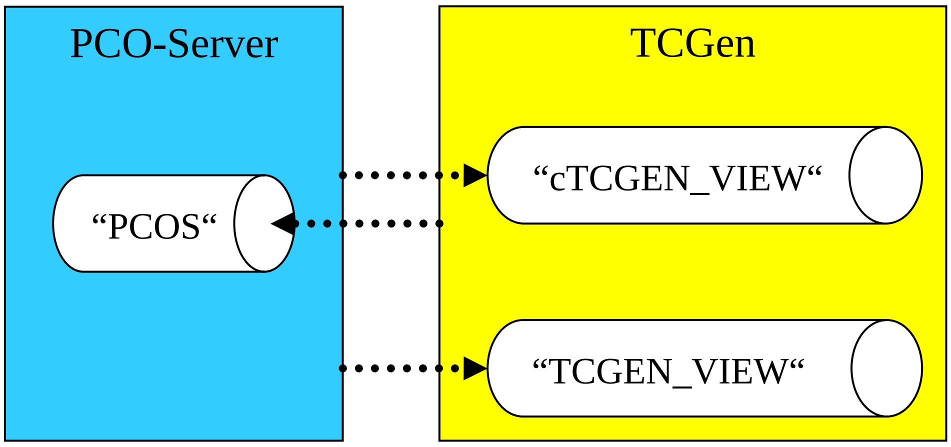


Figure 32 – CMS Queues Used for TCGen-PCO-Communication

In the global main function presented in Source Code 12 on page 92 an instance of `TCGenerator_cmdl` is created, after interpreting command line parameters and the TCGen ini-file. During object construction two CMS queues for communication with PCO-Server are initialized (see Figure 32). This is done actually by the constructor of `PCOView_core` but the names “cTCGEN_VIEW” and “TCGEN_VIEW” for control and data queue are passed to it by TCGen code. This way the automatic naming regarding the current process id of the viewer has been disabled to ease debugging. Furthermore, in the constructor of `TCGenerator_core` the registration at the server is started by using the Viewer Interface function `connect()`. As soon as a response was received, and therefore `on_connected()` is called a filter is set to later receive primitives only. This is sufficient since traces are not considered by the generation algorithm. But, although the server announces complete transmission of all logfile data requested by sending `PCO_LOGFILE_COMPLETE` to TCGen’s control queue, depending on the thread scheduling there might still be some primitives in queue “TCGEN_VIEW” not read out yet. For this reason PCO-Server always sends a final trace carrying the string `"-------- END OF TESTSESSION --------"` after forwarding a recorded session – independent of any filter settings. Thus, the implementation of `on_data()` contains handling for this special trace, too.

After instantiation of the `TCGenerator_cmdl` object the main function passes all parameters gathered from ini-file to it, and depending on the user input at command line the starter method of either analyze- or generate-mode is called. These initialize some protected class members and request the logfile from PCO-Server for the first pass. See, e.g., Source Code 13 on page 93 for the implementation of `generate()`. Now, the main function running in context of the main thread of the TCGen process leaves all further activities to the two threads created by the viewer object, until the latter signals the end of analysis or generation process.

5.5.2 TDC Generation

Triggered by the first data package arriving in the data queue of TCGen and being passed to `on_data()`, the central algorithm described in Subchapter 4.4.2 is started. If the analyze-mode was selected by the user, the functions for writing TDC are skipped and only information about senders, receivers and used SAPs are collected. SAP names can be found out by requesting the name of each primitive from the CCD-Database and considering the first part of it, e.g., “MMGMM” from “MMGMM_REG_REQ”. Since one pass is sufficient to enable a function of `TCGenerator_cmdl` to present the user this information in ASCII-format, the logfile is not re-requested.

The generation-mode is significantly more complex. As a first activity the protected member function `write_initials()` is called inside which all files needed for an TDC test case are created and opened for writing. As alluded in Subchapter 3.5, Test Description Code is typically split into different header and source files representing cases, steps and constraints (see Figure 33).

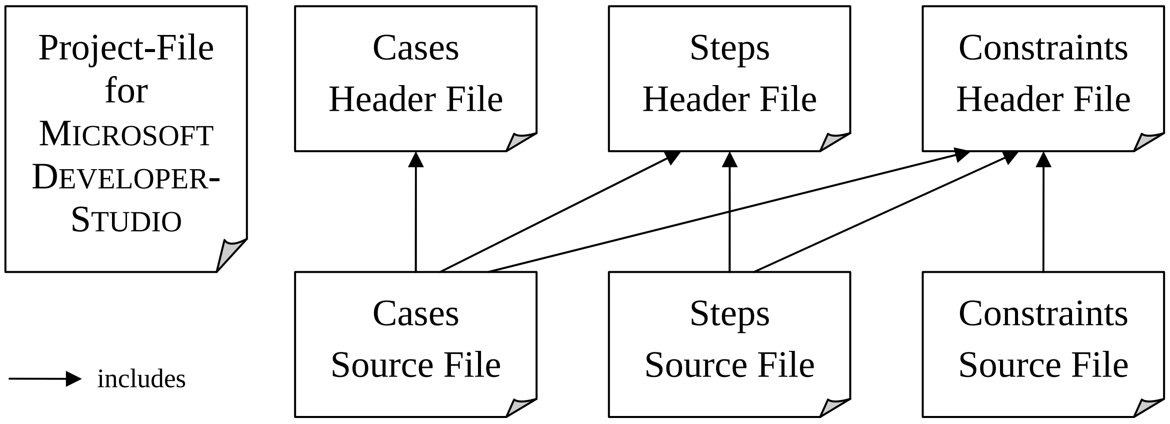
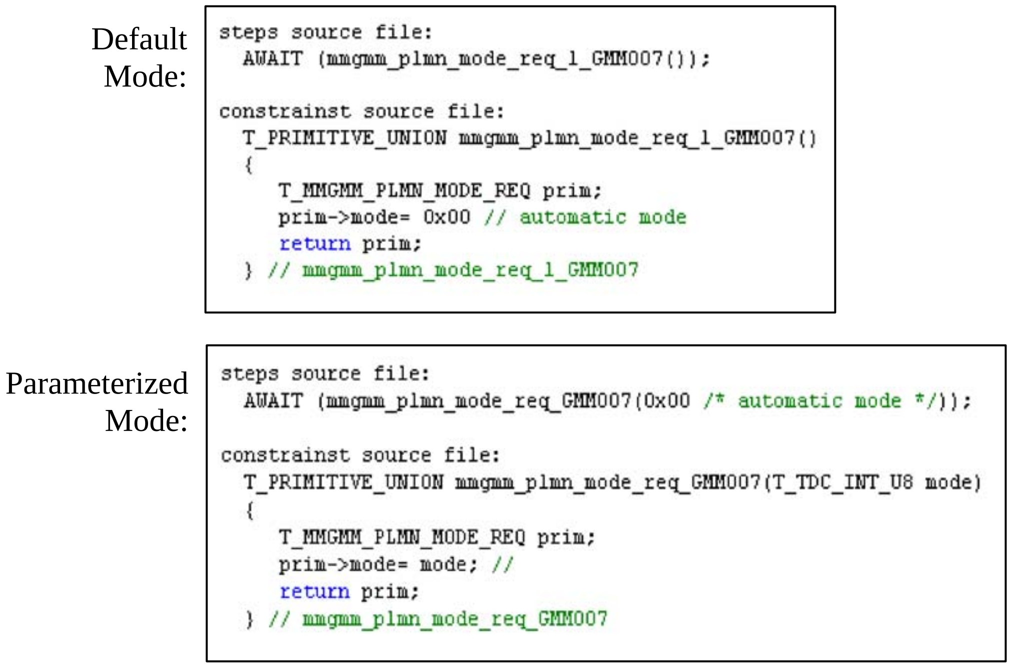


Figure 33 – TDC Files generated by TCGen

Furthermore, a project-file for editing, compilation and linkage in the DEVELOPER-STUDIO from MICROSOFT is produced (see also Subchapter 5.6). This is similar to a makefile containing, e.g., information about external libraries and include paths. The actual values for these have been handed over by the global main function which originally got them from the TCGen ini-file. In Source Code 10 on page 91 the default version of such a file can be looked up. The source file containing the TDC-`CASE` can also be created at once. Calls of two TDC-`STEPS` are inserted into the new `CASE`: one for setting up routing and another for the input/output specifications. Some initial lines of code are written to the other TDC files, as well. Beside various comments explaining their content and stating, e.g., the PCO logfile used as input, the Entity Graph is utilized to generate the routing commands inside the first `STEP`, based on the list of entities specified by the user which shall be tested as a block. The beginning sequence for the second `STEP` is also written to the appropriate file.

Unfortunately, no information about SAP names can be gathered beforehand. For that reason, the first pass applied to the logfile data has to be “wasted” to create the `#include`-statements for further header files. These are files like `p_MMGMM.h` or `m_SM.h` containing C++-declarations of primitive and air-message structures. They have been created together with the CCD-Database-DLL (see Figure 21, page 51) but cannot be used internally by TCGen; this would require recompilation of the tool each time an interface has changed. For the compilation of test cases they are indispensable, but only those are needed, whose structures are actually used. By extracting the SAP parts from the primitive names as explained above the `p_*`-files can easily be identified. To find out the required `m_*`-files each primitive has to be checked for carried air-messages.

After the special trace which marks the end of data transmission has been received, the protected member variable `m_pass_nr` is increased, and the PCO-Server is contacted to send the logged primitives again. This process will be repeated until the last TDC-constraint is generated. During each pass declarations in header files and corresponding definitions are created in parallel. To retrieve the needed information from the CCD-Database an initial pointer to the structure table for each primitive is requested, which will be passed to function `write_params()` with parameter `level` set to `0`. This function reads out and examines the elements of the current level and, in case of substructures, recursively calls itself with increased `level`. By considering `m_pass_nr` and `level` the decision is made, whether a TDC constraint shall be written for the processed element. For elements with basic type (see Source Code 14, page 94) the equation `level==m_pass_nr-1` has to be true. The same holds for creating substructure assignments, but `level==m_pass_nr-2` is the trigger for writing begin and end sequences of structure definitions (see Source Code 15, page 95). This way, constraints for all elements can be generated sequentially and level by level. Arrays and air-messages, which have to be decoded before further processing, are handled in a similar manner.


Default Mode:

```
steps source file:
  AWAIT (mmgmm_plmn_mode_req_1_GMM007());

constrainst source file:
  T_PRIMITIVE_UNION mmgmm_plmn_mode_req_1_GMM007()
  {
     T_MMGMM_PLMN_MODE_REQ prim;
     prim->mode= 0x00 // automatic mode
     return prim;
  } // mmgmm_plmn_mode_req_1_GMM007
```

Parameterized Mode:

```
steps source file:
  AWAIT (mmgmm_plmn_mode_req_GMM007(0x00 /* automatic mode */));

constrainst source file:
  T_PRIMITIVE_UNION mmgmm_plmn_mode_req_GMM007(T_TDC_INT_U8 mode)
  {
     T_MMGMM_PLMN_MODE_REQ prim;
     prim->mode= mode; //
     return prim;
  } // mmgmm_plmn_mode_req_GMM007
```

Figure 34 – Comparison of Default and Parameterized Mode

During the first none-initial pass the `SEND` and `AWAIT` commands are additionally written into the steps source file. They get primitive constraints as parameters. Per default one constraint is created for each primitive found, no matter whether there are some of the same type but with different element values. In the so-called parameterized mode which can be selected in the ini-file, TCGen is able to generate only one constraint for each primitive type but with parameters corresponding to the elements at level 0. The actual values are forwarded to the constraints inside the `SEND`/`AWAIT` commands (see Figure 34). Of course, this mode has to be taken into account inside `write_params()`, too. The variables `m_parameters` and `m_header_creation` are used in this connection. The general naming scheme is also affected: Usually constraints are labelled like the primitive/substructure elements, suffixed by a continuously increased number. In parameterized mode no such number is necessary for primitive constraints. However, the additional suffix derived from the name of the test case currently generated is appended in either case. It allows TCGen to incrementally add test cases to the same TDC files by keeping cases, steps and constraints with different final suffixes. If the same test case name is specified by the user in a subsequent execution of TCGen, formerly generated constructs are overwritten.

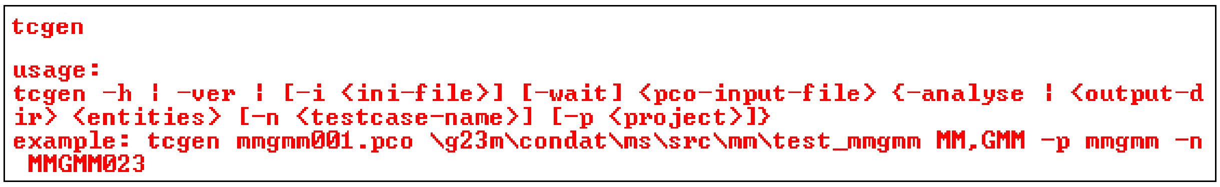
Concerning the rule processing capabilities a dedicated class `Rules` has been implemented which encapsulates all necessary routines. It is derived from class `XmlNotify` (see Figure 31, page 64) whose source code is freely available and which provides methods for parsing XML-files. The latter are needed because the author decided to use this format for rule files. It supports parameterized tags, optionally containing values, which exactly match the needs to express rules for the test case generator, e.g.:

```
<change primitive="MMGMM_*" param="detach_cause">
     (%v>=1 ? 0x5 : %v) </change>
```

More examples can be found in Source Code 17 on page 97. See [XML] for a comprehensive introduction in XML. The implementation of class `Rules` is straight forward (see Source Code 16, page 96 for its declaration): While reading the XML-file, fields of dedicated structures for the skip- and change-rules are filled. Whenever a primitive or element (parameter) name is passed to one of the `skip_*()` or `change_*()` functions, these fields are scanned for matching entries and appropriate values are returned. Skipping is then realized by inserting C++ comment identifiers (“`//`”) before `SEND/AWAIT`-commands, respectively element assignments. This is possible because un-initialized elements are skipped automatically in TDC; and by doing so the user can easily re-enable the assignment if needed. The algorithm used for the changing of values is presented in Source Code 18 on page 96.

5.5.3 *User Interface*

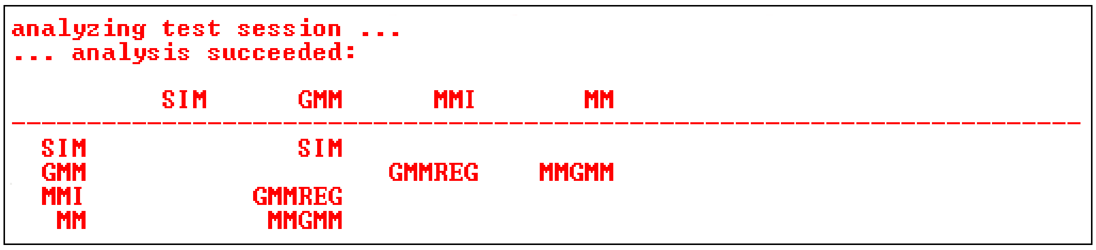
The user interface of a test tool is very important for its acceptance by testers. It should respond with adequate performance and be intuitively to handle. TCGen currently supports a command line interface only, whose parameters are listed if the executable is called without any of them (see Figure 35).



```
tcgen

usage:
tcgen -h | -ver | [-i <ini-file>] [-wait] <pco-input-file> {-analyse | <output-d
ir> <entities> [-n <testcase-name>] [-p <project>]}
example: tcgen mmgmm001.pco \g23m\condat\ms\src\mm\test_mmgmm MM,GMM -p mmgmm -n
 MMGMM023
```

Figure 35 – Command Line Parameters of TCGen

Beside the optional name of an ini-file which shall be used instead of the default one, at minimum a PCO logfile has to be specified. If additionally the analyze-mode is chosen output similar to the one in Figure 36 will be produced. Such a table is useful to get a quick overview about the content of a logfile regarding the recorded communication.

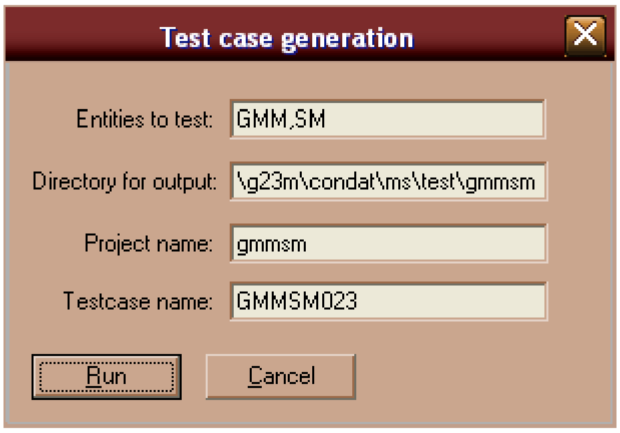


```
analyzing test session ...
... analysis succeeded:

          SIM      GMM      MMI       MM
-----------------------------------------------------------------------
  SIM              SIM
  GMM                    GMMREG    MMGMM
  MMI           GMMREG
   MM            MMGMM
```

Figure 36 – Output of TCGen Analyze Mode

For the generation process some more parameters are required: a directory where the TDC files shall be stored, or where already existing can be opened to add a test case; a comma separated list of entity names which will be considered as the entities under test; a name for the project file which will also be the base name for the other TDC files; and finally a name for the test case. TCGen has been tested with logfiles containing up to 100 primitives and it took the tool only some seconds to complete a generation. This is much less time then the subsequent compilation process will need to finish (see Subchapter 5.6).

The described possibilities are sufficient to use all capabilities of TCGen and the final output of the generator contains an example for compiling the test case from command line. Nevertheless, many users will not be satisfied by the described interface. They expect a GUI from today's tools where all parameters can be edited inside sensibly designed dialogs, including, of course, the ini-file content and rules. Moreover, in an optimal way after pressing only one button all necessary processes should be started automatically. Surely, not all user fantasies are reasonable or even possible to be realized. But a first step was made by including a dedicated dialog into the GUI version of PCO-Controller as shown in Figure 37. This allows the input of TCGen's command line parameters, which will be forwarded together with the logfile selected in the main dialog of the controller to the test generator executable. Another advantage of this constellation: The PCO-Server is started automatically.



Entities to test: GMM,SM
Directory for output: \g23m\condat\ms\test\gmmsm
Project name: gmmsm
Testcase name: GMMSM023

Figure 37 – PCO Dialog to Call TCGen

5.6 Test Case Compilation and Execution

Before a TDC test case – either written manually or generated by TCGen – can be executed, it has to be converted into a WINDOWS-DLL which is then loaded by the TAP tool. Since TDC is actually a subset of the C++ language the conversion is just a usual compilation, followed by the linkage with TDC core objects. The actual execution process takes place when TAP calls the specified send/await constructs and evaluates using the CCD-Database. Together with the optionally started PCO results are created (see Figure 38).

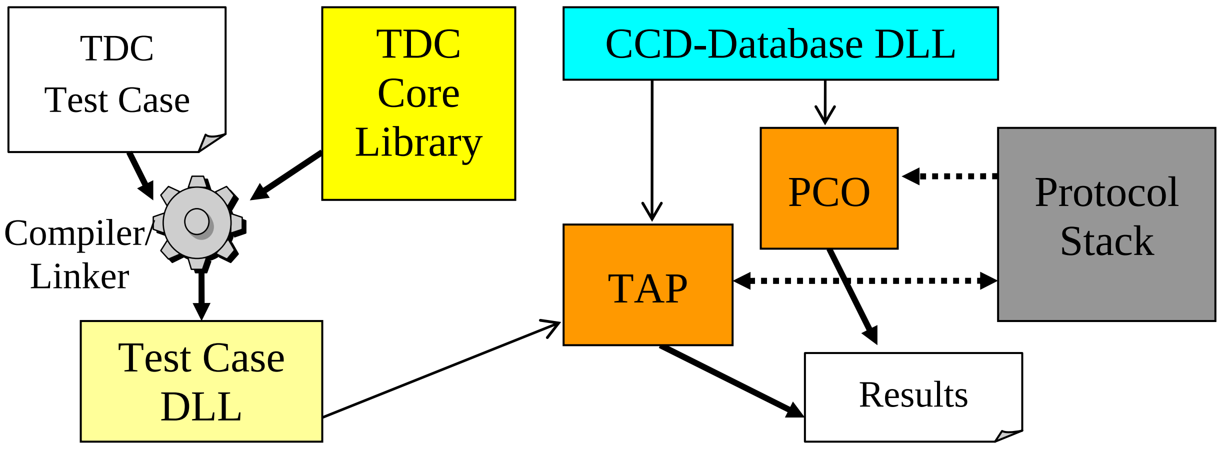


Figure 38 – Creation and Execution of a Test Case DLL

To make the process of starting TAP, PCO and the PS simulation more transparent for the user, a GUI application called *TAP-Caller* exists, which has been partly implemented by the author. Figure 39 gives an impression of its interface. TAP-Caller is also capable to generate reports for a set of executed test cases.

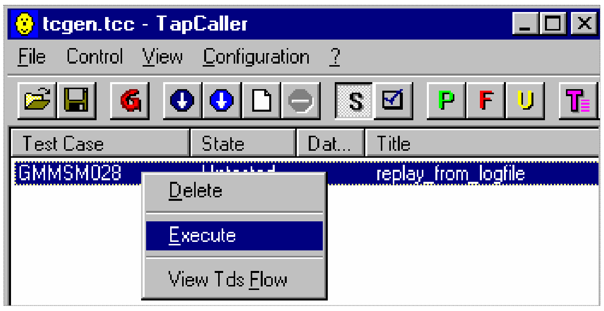


Figure 39 – TAP-Caller Interface

5.7 Comparison with Other Methods

Comparing TCGen with other generators is rather difficult for the following reasons: First, no test case generator has actually been used at TEXAS INSTRUMENTS BERLIN so far. The only attempt made with the commercial tool from TESTINGTECH (see [TESTTECH]) proved after more then a years adaptation work by this company that it is, in principle, possible to create and run test cases inside their framework, which has to connect to the Test Interface. Beside the fact that no special framework adaptations are necessary for TCGen, the generator included in the TESTINGTECH product takes SDL/MSC specifications as input. Indeed, no generator known to the author uses logged data as TCGen does. There exist several replay tools which can reproduce the data collected during test runs, but they are no real generators whose output can be used, e.g., together with a protocol stack simulation. Hence, the only meaningful comparison can be done with manual test case writing.

At the company the author is currently working test cases are usually written by developers familiar with specific details of the entities they are maintaining. In Figure 40 the improvement they could gain from TCGen is estimated:

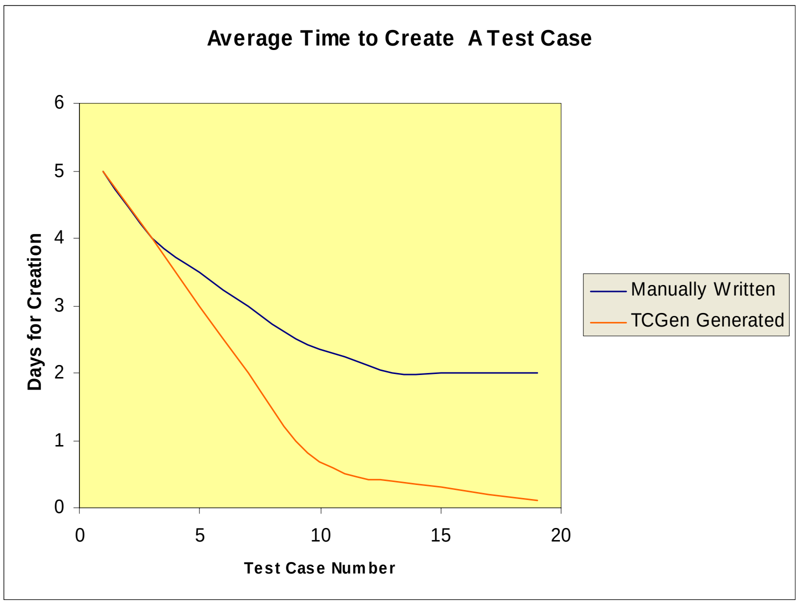


Figure 40 – Estimated Speed Up of Test Case Creation

This estimation is based on the assumption that PCO logfiles are recorded in parallel by other persons, which usually is the case. In both situations the creation of first test cases takes a quite long time – for manually writing because people have to get used to the test script language and while using TCGen because initially no rule files exist. After this consolidation phase the developers interviewed by the author estimate the effort per test case to one to three days. If TCGen would be used together with rules tuned for the specific needs more then one case could be produced during a day, with the additional advantage that generation takes place in background allowing other activities in parallel. While no significant further performance boost can be expected in the manual case, continuously improvement of TCGen rule files will speed up the process even more. Of course, this calculations apply only to the generation of test cases which don't need to be adapted or reworked, e.g., to regression tests of already functioning behaviors. But for many lines of work in Berlin the design and initial implementation phases are over already and regression tests of this kind are highly demanded, as well as the reproduction of bugs happened in field tests.

In addition, TCGen offers a feature which is quite difficult to achieve by manually writing: several entities can be considered as a block and be tested together in so-called multi-entity test cases. For example, this option is very important if, as it is currently the case with development for UMTS and GPRS, different implementations of the same entities exist, which can only be considered as one construct until the developments have been aligned. However, automatic generation will not completely substitute manual processing as long as no information from specifications is used, too. And even if that would be the case, testers and developers typically want to keep the possibility to easily edit/adapt certain parts of a test case. TCGen supports this by generating human readable compositions as far as possible.

5.8 Testing the Test Tools

A common problem appearing with tools for testing other software is the fact that the tools themselves are not sufficiently tested. Usually, after a certain period of suspicious trial-usage followed by a time of heavier utilization at least proving acceptable stability, the tool is classified as reliable. The speed of this process also depends on the availability of alternatives. To improve the procedure of getting feedback from users and fixing bugs or implementing new features regarding this input, the author invented a utility called Moan-Button which consists of a DLL optionally linked to an application, and providing a small hammer icon in the *system tray*. Clicking on it enables the user to directly send an email to the current maintainer of the running tool. Of course, especially in the initial development phase various tests have to be done by the developer himself and the user should be considered as the last instance for finding errors. But unless formal correctness has been proved no complex test tool can be delivered without bugs.

As an example, PCO has been evolved in the way described, but although automating test case generation is highly demanded, it was rather hard to find beta-testers for first pre-releases of TCGen. The main reason was probably that time is extremely valuable for developers in the mobile business, and they don‘t want to waste hours by trying out tools whose output might not be usable. On the other hand the author could ensure suitable stability and reliability quite from the first versions of TCGen. This is because of several reasons: First of all, the interaction with the framework has been derived from PCO via the Viewer Interface and the utilization of the CCD-Database to, e.g., interpret primitive structures is implemented in a very similar way in PCO and TAP. Both tools have been used successfully for a long time which, of course, does not guarantee error-freeness. Furthermore, dedicated tests have been and are applied on the TCGen application. One of the methods used is presented in Figure 41. Based on existing test cases logfiles are created during TAP execution. These are provided as input to the test case generator, instead of field test recordings.

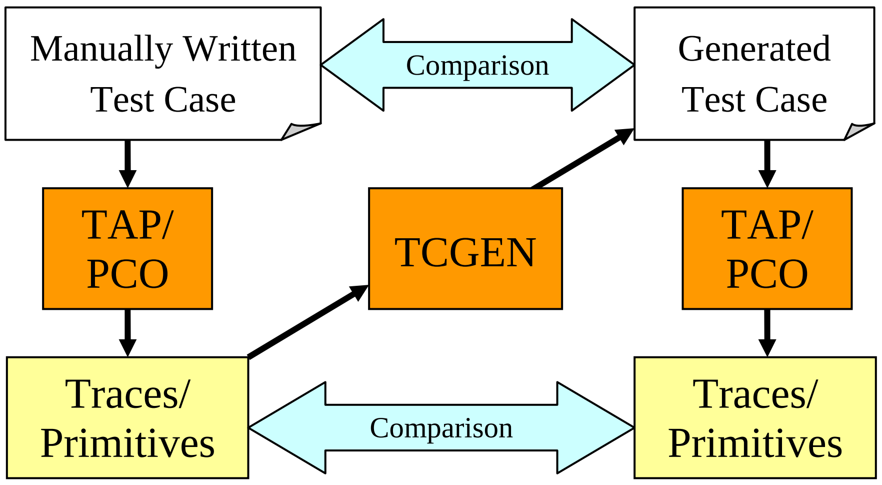


Figure 41 – Method of Testing TCGen

To enable TCGen to create new test cases out of such logfiles, the generation algorithm had to be enhanced by a special handling of the entity TAP, which now appears as sender of some primitives. This enhancement is kept in official releases which makes sense not only to exclude untested side-effects. It can also be used to convert test cases written in an older format still understood by TAP into TDC scripts. For TCGen-testing purposes the resulting TDC files are compared with the original test case, which unfortunately is hard to automate since TDC is very flexible. Thus, a more efficient technique is to re-run TAP with the generated test case and thereafter compare the traces and primitives.

To check the rule processing a special XML-file is maintained, containing a set of rules which covers all equivalency classes (e.g., “with wildcards”, “without wildcards”, …) identified by the author. The current beta-release of TCGen has successfully passed all tests. Nevertheless, there are still many things to improve and probably some minor bugs not yet detected.

6 Conclusions

The aims of this diploma thesis were to:

– give a general overview about the state of the art in software testing

– present specifics concerning software and testing in the mobile sector

– introduce the tracing mechanism used at TEXAS INSTRUMENTS BERLIN AG

– explain the theoretical concepts underlying the test case generator designed and developed by the author

– give details concerning the actual implementation of this generator

The author tried to summarize facts concerning software testing methods and explain their importance for the mobile business as well as requirements arising in this particular sector. Despite increased efforts to improve the design process (such as “clean-room software engineering”) for the products have fewer defects from the start – as this is expected to be more efficient than testing – testing is currently the main tool to ensure quality and stability.

Furthermore, automating as much parts of the testing process as possible is one of the main issues in these days. The test case generator presented in this document represents one possibility to come closer to that goal regarding the creation of regression tests and the reproduction of erroneous behavior. It strongly depends on the tools PCO and TAP, as well as on the frameworks which have been examined in this paper, too. Currently, output can be created in the TDC format only. However, great efforts have been made to keep the functionalities as general as possible, and TTCN-3 will probably be the next format supported. The long term planning even includes the idea of passing the BNF of any favored test language to TCGen. Utilization of other input beside the PCO logfiles, e.g., specifications is another option which might be considered.

The main problems which have to be solved presently are related to the tracing mechanism, the essential way to get useful information from a running protocol stack. Not every interface between two entities is realized via primitive exchange and can be observed. Moreover, the data rates available for the Test Interface are too low to allow duplication of the complete communication flow. Finally, the more information is traced out the higher the CPU load will be inside the mobile device or test board. Nevertheless, the generation of much more test cases then before is already possible with TCGen. Especially in the beginning many of them will contain redundant parts, but the code coverage can be increased significantly – one main request from the TEXAS INSTRUMENTS management.

As mentioned in Chapter 5, the test applications support only MS WINDOWS at this time. To provide versions for Linux, e.g., by using a portable GUI library like *QT* from TROLLTECH (see [TROLL]), will be another important task for the future as already many computers in testing laboratories use *UNIX*-like operating systems. Another approach would be to use JAVA (see [JAVA], [JAVABOOK]) for further developments as it becomes increasingly more common in the mobile sector.

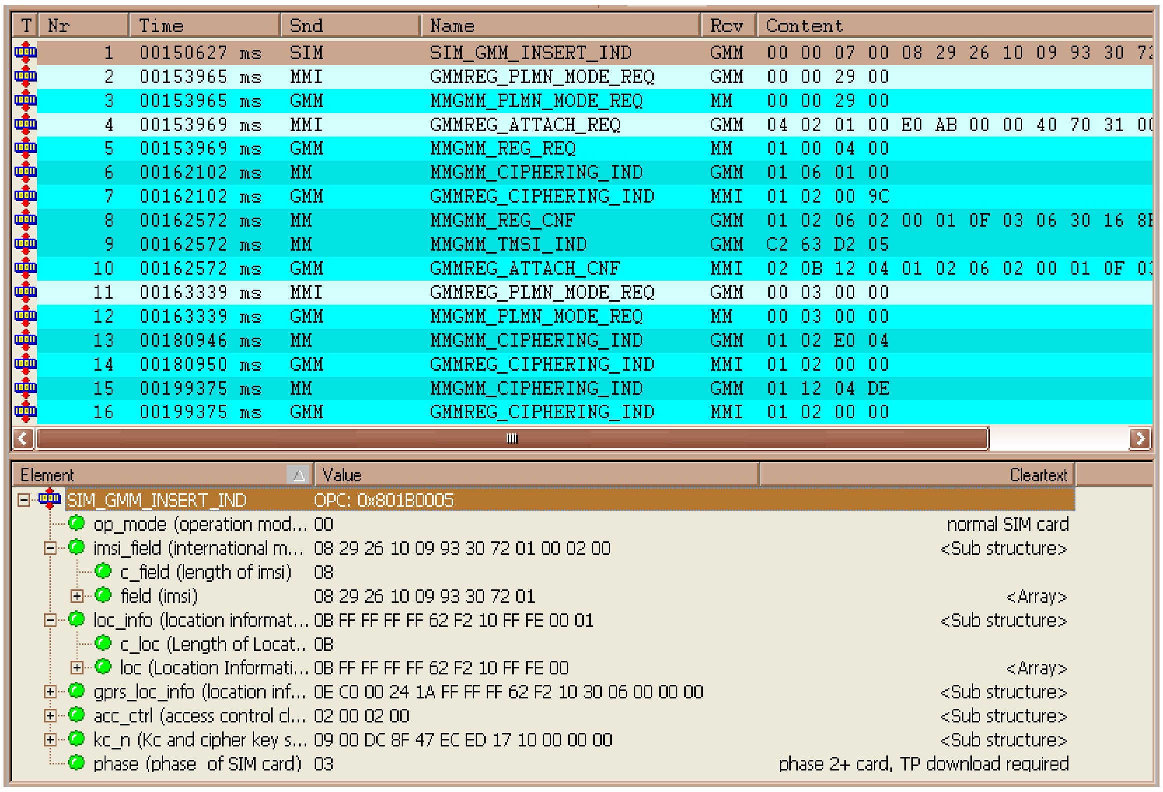
However, a first functional solution has been provided and now users have to be found to get feedback for improvements regarding the concrete needs of workers at the company. Some issues for the near future are already registered: the support of wildcards in rules has to be extended; structures of the same type should be united where possible, even among different test cases to increase readability; constants declared in the SAP header files shall be used instead of actual values; and finally, the CCD-Database used during the logging process should be included in the PCO logfile to avoid a later search for the right DLL. The author is ready for the challenges to come.

Appendix

A.1 *Example of a Generated Test Case*

The following figures document the generation of a test case out of data recorded during the input of a *PIN* at a cellular phone. GMM and MM build up the observed and tested entity-block.

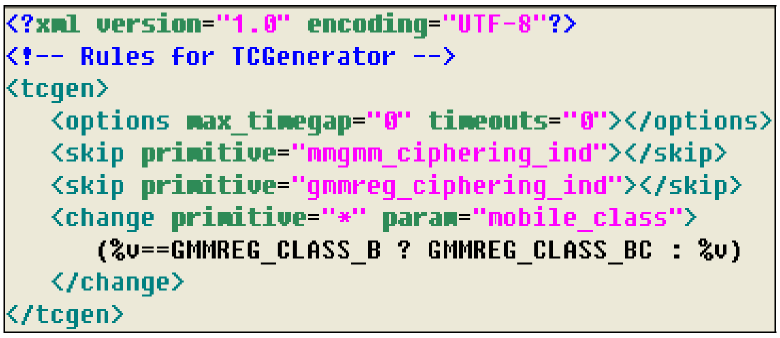
Received primitives displayed and logged by the PCO tool:



T	Nr	Time	Snd	Name	Rcv	Content
	1	00150627 ms	SIM	SIM_GMM_INSERT_IND	GMM	00 00 07 00 08 29 26 10 09 93 30 7
	2	00153965 ms	MMI	GMMREG_PLMN_MODE_REQ	GMM	00 00 29 00
	3	00153965 ms	GMM	MMGMM_PLMN_MODE_REQ	MM	00 00 29 00
	4	00153969 ms	MMI	GMMREG_ATTACH_REQ	GMM	04 02 01 00 E0 AB 00 00 40 70 31 0
	5	00153969 ms	GMM	MMGMM_REG_REQ	MM	01 00 04 00
	6	00162102 ms	MM	MMGMM_CIPHERING_IND	GMM	01 06 01 00
	7	00162102 ms	GMM	GMMREG_CIPHERING_IND	MMI	01 02 00 9C
	8	00162572 ms	MM	MMGMM_REG_CNF	GMM	01 02 06 02 00 01 0F 03 06 30 16 8
	9	00162572 ms	MM	MMGMM_TMSI_IND	GMM	C2 63 D2 05
	10	00162572 ms	GMM	GMMREG_ATTACH_CNF	MMI	02 0B 12 04 01 02 06 02 00 01 0F 0
	11	00163339 ms	MMI	GMMREG_PLMN_MODE_REQ	GMM	00 03 00 00
	12	00163339 ms	GMM	MMGMM_PLMN_MODE_REQ	MM	00 03 00 00
	13	00180946 ms	MM	MMGMM_CIPHERING_IND	GMM	01 02 E0 04
	14	00180950 ms	GMM	GMMREG_CIPHERING_IND	MMI	01 02 00 00
	15	00199375 ms	MM	MMGMM_CIPHERING_IND	GMM	01 12 04 DE
	16	00199375 ms	GMM	GMMREG_CIPHERING_IND	MMI	01 02 00 00

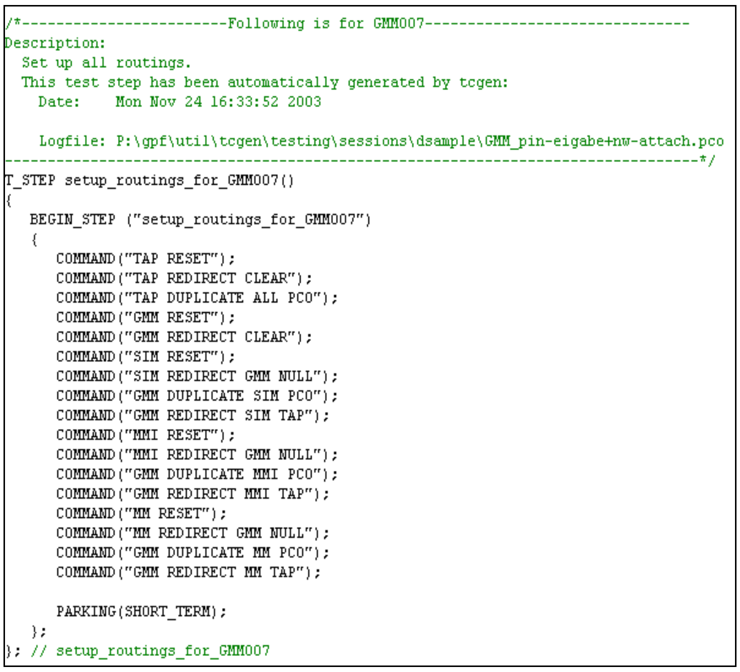
Element	Value	Cleartext
SIM_GMM_INSERT_IND	OPC: 0x801B0005	
op_mode (operation mod...	00	normal SIM card
imsi_field (international m...	08 29 26 10 09 93 30 72 01 00 02 00	<Sub structure>
c_field (length of imsi)	08	
field (imsi)	08 29 26 10 09 93 30 72 01	<Array>
loc_info (location informat...	0B FF FF FF FF 62 F2 10 FF FE 00 01	<Sub structure>
c_loc (Length of Locat..	0B	
loc (Location Informati...	0B FF FF FF FF 62 F2 10 FF FE 00	<Array>
gprs_loc_info (location inf...	0E C0 00 24 1A FF FF FF 62 F2 10 30 06 00 00 00	<Sub structure>
acc_ctrl (access control cl...	02 00 02 00	<Sub structure>
kc_n (Kc and cipher key s...	09 00 DC 8F 47 EC ED 17 10 00 00 00	<Sub structure>
phase (phase of SIM card)	03	phase 2+ card, TP download required

Rules file for TCGen with, e.g., modification of the expected mobile class:



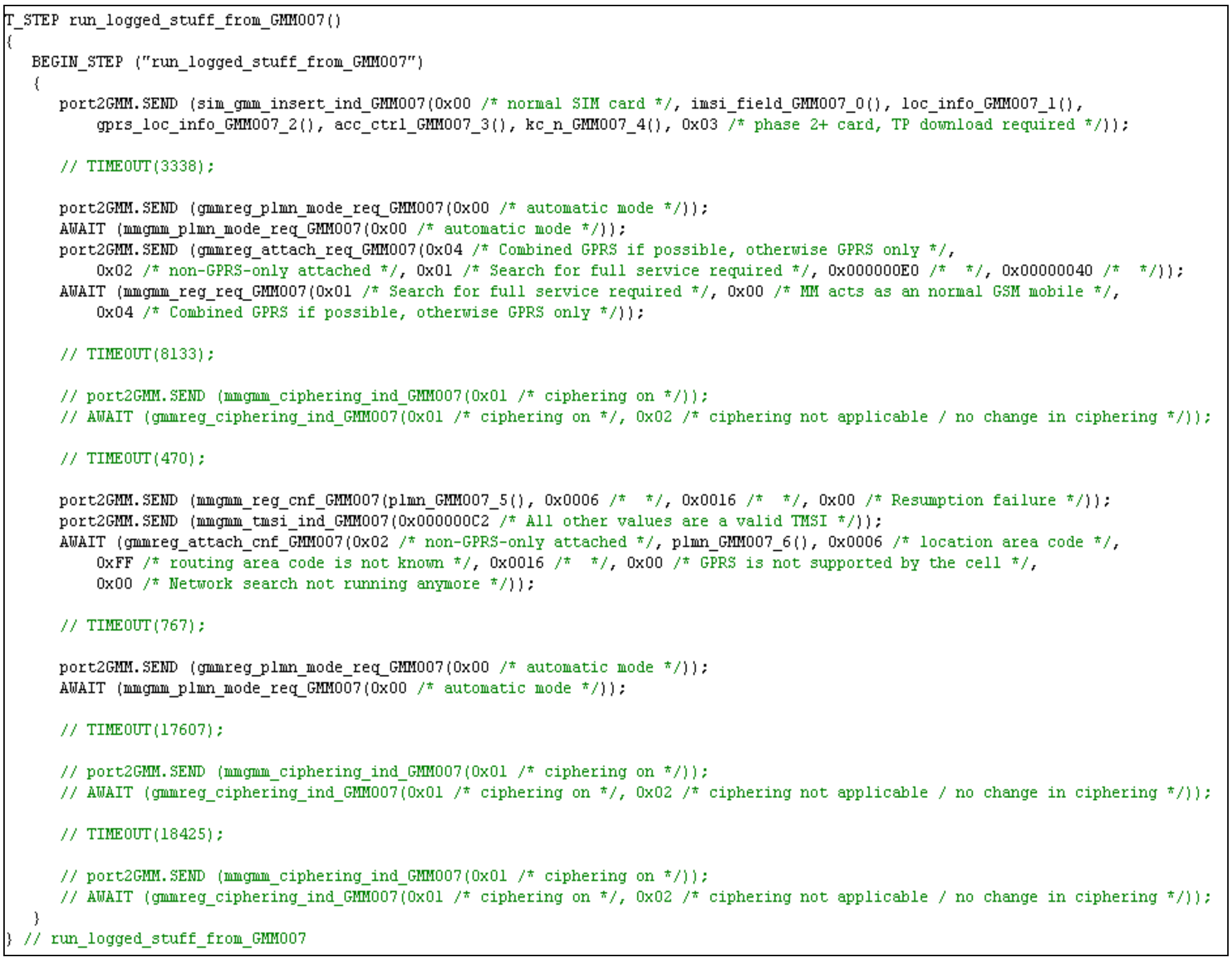
```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Rules for TCGenerator -->
<tcgen>
   <options max_timegap="0" timeouts="0"></options>
   <skip primitive="mmgmm_ciphering_ind"></skip>
   <skip primitive="gmmreg_ciphering_ind"></skip>
   <change primitive="*" param="mobile_class">
      (%v==GMMREG_CLASS_B ? GMMREG_CLASS_BC : %v)
   </change>
</tcgen>
```

TDC step for setting up routings needed:



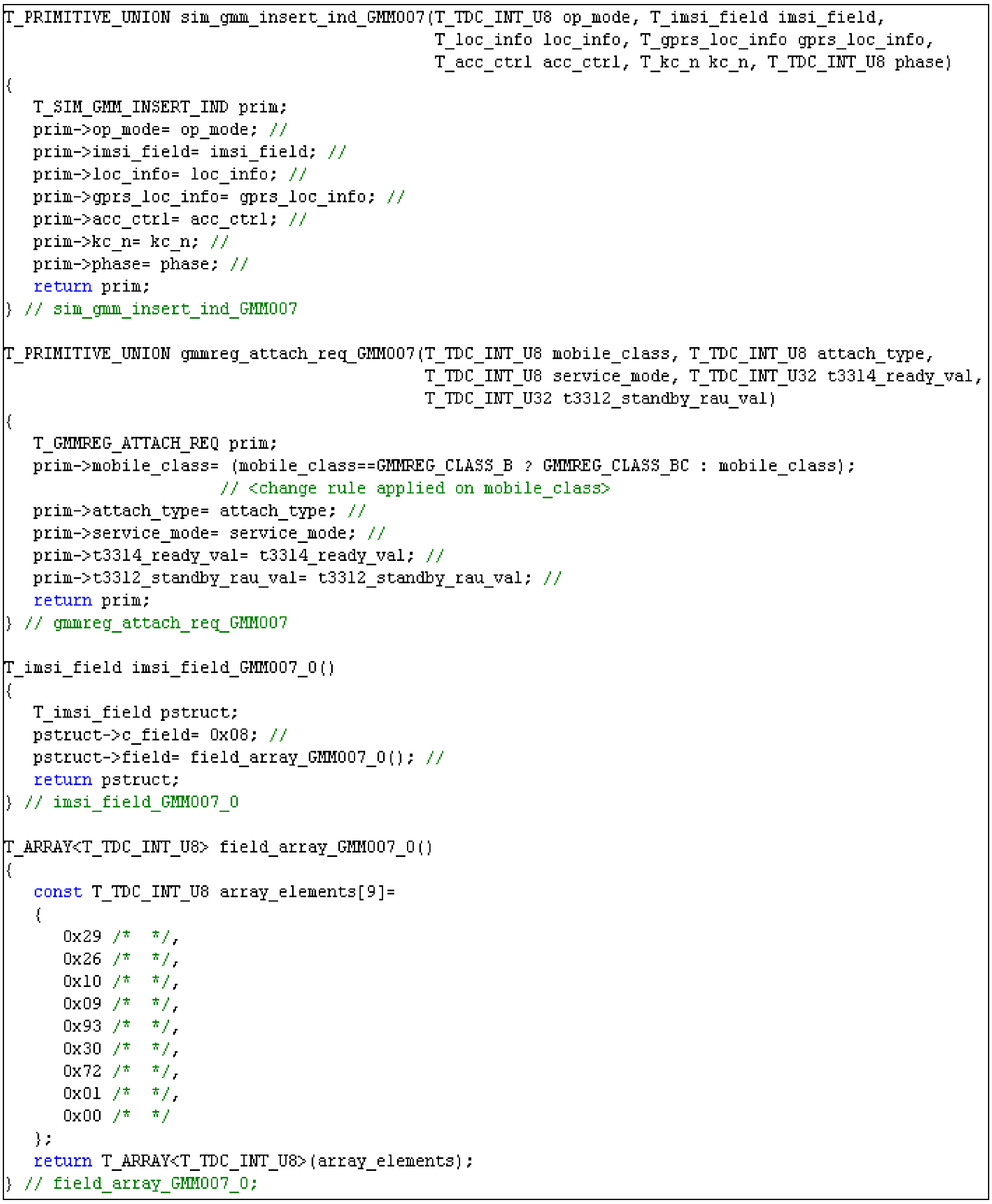
```
/*------------------------Following is for GMM007-------------------------------
Description:
  Set up all routings.
  This test step has been automatically generated by tcgen:
    Date:    Mon Nov 24 16:33:52 2003

    Logfile: P:\gpf\util\tcgen\testing\sessions\dsample\GMM_pin-eigabe+nw-attach.pco
---------------------------------------------------------------------------------*/
T_STEP setup_routings_for_GMM007()
{
   BEGIN_STEP ("setup_routings_for_GMM007")
   {
      COMMAND("TAP RESET");
      COMMAND("TAP REDIRECT CLEAR");
      COMMAND("TAP DUPLICATE ALL PCO");
      COMMAND("GMM RESET");
      COMMAND("GMM REDIRECT CLEAR");
      COMMAND("SIM RESET");
      COMMAND("SIM REDIRECT GMM NULL");
      COMMAND("GMM DUPLICATE SIM PCO");
      COMMAND("GMM REDIRECT SIM TAP");
      COMMAND("MMI RESET");
      COMMAND("MMI REDIRECT GMM NULL");
      COMMAND("GMM DUPLICATE MMI PCO");
      COMMAND("GMM REDIRECT MMI TAP");
      COMMAND("MM RESET");
      COMMAND("MM REDIRECT GMM NULL");
      COMMAND("GMM DUPLICATE MM PCO");
      COMMAND("GMM REDIRECT MM TAP");

      PARKING(SHORT_TERM);
   };
}; // setup_routings_for_GMM007
```

Main TDC step containing send/await commands:



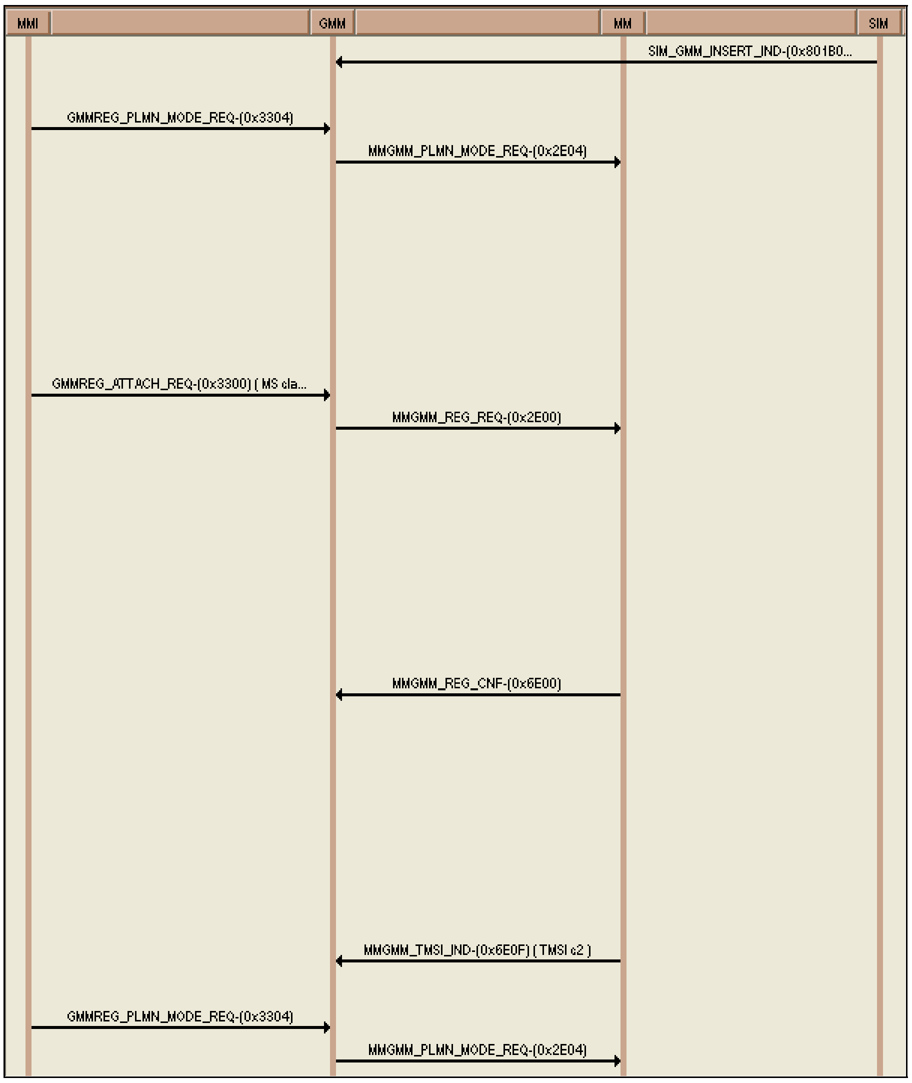
```
T_STEP run_logged_stuff_from_GMM007()
{
   BEGIN_STEP ("run_logged_stuff_from_GMM007")
   {
      port2GMM.SEND (sim_gmm_insert_ind_GMM007(0x00 /* normal SIM card */, imsi_field_GMM007_0(), loc_info_GMM007_1(),
          gprs_loc_info_GMM007_2(), acc_ctrl_GMM007_3(), kc_n_GMM007_4(), 0x03 /* phase 2+ card, TP download required */));

      // TIMEOUT(3338);

      port2GMM.SEND (gmmreg_plmn_mode_req_GMM007(0x00 /* automatic mode */));
      AWAIT (mmgmm_plmn_mode_req_GMM007(0x00 /* automatic mode */));
      port2GMM.SEND (gmmreg_attach_req_GMM007(0x04 /* Combined GPRS if possible, otherwise GPRS only */,
          0x02 /* non-GPRS-only attached */, 0x01 /* Search for full service required */, 0x000000E0 /*  */, 0x00000040 /*  */));
      AWAIT (mmgmm_reg_req_GMM007(0x01 /* Search for full service required */, 0x00 /* MM acts as an normal GSM mobile */,
          0x04 /* Combined GPRS if possible, otherwise GPRS only */));

      // TIMEOUT(8133);

      // port2GMM.SEND (mmgmm_ciphering_ind_GMM007(0x01 /* ciphering on */));
      // AWAIT (gmmreg_ciphering_ind_GMM007(0x01 /* ciphering on */, 0x02 /* ciphering not applicable / no change in ciphering */));

      // TIMEOUT(470);

      port2GMM.SEND (mmgmm_reg_cnf_GMM007(plmn_GMM007_5(), 0x0006 /*  */, 0x0016 /*  */, 0x00 /* Resumption failure */));
      port2GMM.SEND (mmgmm_tmsi_ind_GMM007(0x000000C2 /* All other values are a valid TMSI */));
      AWAIT (gmmreg_attach_cnf_GMM007(0x02 /* non-GPRS-only attached */, plmn_GMM007_6(), 0x0006 /* location area code */,
          0xFF /* routing area code is not known */, 0x0016 /*  */, 0x00 /* GPRS is not supported by the cell */,
          0x00 /* Network search not running anymore */));

      // TIMEOUT(767);

      port2GMM.SEND (gmmreg_plmn_mode_req_GMM007(0x00 /* automatic mode */));
      AWAIT (mmgmm_plmn_mode_req_GMM007(0x00 /* automatic mode */));

      // TIMEOUT(17607);

      // port2GMM.SEND (mmgmm_ciphering_ind_GMM007(0x01 /* ciphering on */));
      // AWAIT (gmmreg_ciphering_ind_GMM007(0x01 /* ciphering on */, 0x02 /* ciphering not applicable / no change in ciphering */));

      // TIMEOUT(18425);

      // port2GMM.SEND (mmgmm_ciphering_ind_GMM007(0x01 /* ciphering on */));
      // AWAIT (gmmreg_ciphering_ind_GMM007(0x01 /* ciphering on */, 0x02 /* ciphering not applicable / no change in ciphering */));
   }
} // run_logged_stuff_from_GMM007
```

Part of the TDC structure definitions:

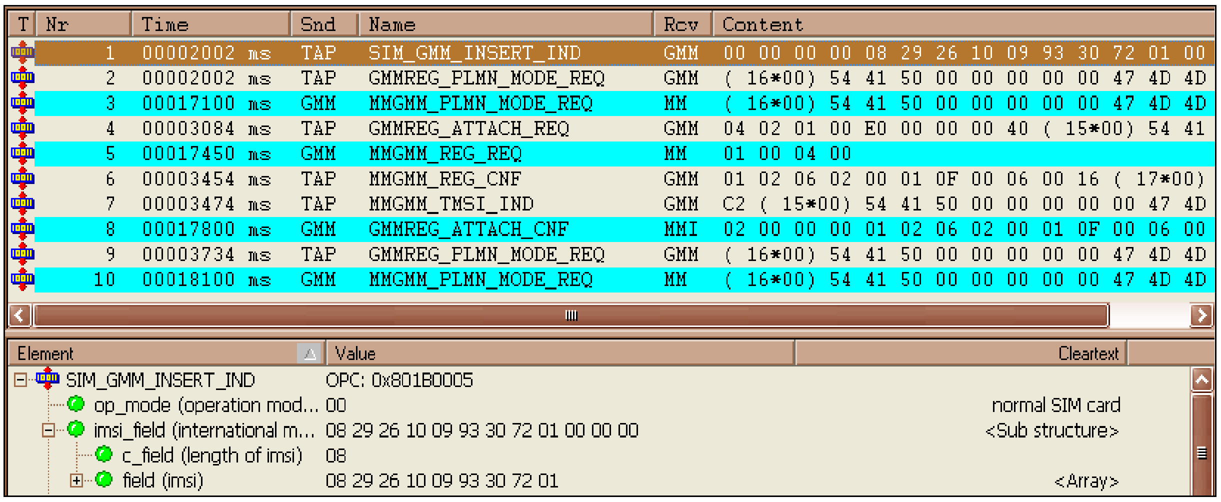


```
T_PRIMITIVE_UNION sim_gmm_insert_ind_GMM007(T_TDC_INT_U8 op_mode, T_imsi_field imsi_field,
                                            T_loc_info loc_info, T_gprs_loc_info gprs_loc_info,
                                            T_acc_ctrl acc_ctrl, T_kc_n kc_n, T_TDC_INT_U8 phase)
{
   T_SIM_GMM_INSERT_IND prim;
   prim->op_mode= op_mode; //
   prim->imsi_field= imsi_field; //
   prim->loc_info= loc_info; //
   prim->gprs_loc_info= gprs_loc_info; //
   prim->acc_ctrl= acc_ctrl; //
   prim->kc_n= kc_n; //
   prim->phase= phase; //
   return prim;
} // sim_gmm_insert_ind_GMM007

T_PRIMITIVE_UNION gmmreg_attach_req_GMM007(T_TDC_INT_U8 mobile_class, T_TDC_INT_U8 attach_type,
                                           T_TDC_INT_U8 service_mode, T_TDC_INT_U32 t3314_ready_val,
                                           T_TDC_INT_U32 t3312_standby_rau_val)
{
   T_GMMREG_ATTACH_REQ prim;
   prim->mobile_class= (mobile_class==GMMREG_CLASS_B ? GMMREG_CLASS_BC : mobile_class);
                     // <change rule applied on mobile_class>
   prim->attach_type= attach_type; //
   prim->service_mode= service_mode; //
   prim->t3314_ready_val= t3314_ready_val; //
   prim->t3312_standby_rau_val= t3312_standby_rau_val; //
   return prim;
} // gmmreg_attach_req_GMM007

T_imsi_field imsi_field_GMM007_0()
{
   T_imsi_field pstruct;
   pstruct->c_field= 0x08; //
   pstruct->field= field_array_GMM007_0(); //
   return pstruct;
} // imsi_field_GMM007_0

T_ARRAY<T_TDC_INT_U8> field_array_GMM007_0()
{
   const T_TDC_INT_U8 array_elements[9]=
   {
      0x29 /*  */,
      0x26 /*  */,
      0x10 /*  */,
      0x09 /*  */,
      0x93 /*  */,
      0x30 /*  */,
      0x72 /*  */,
      0x01 /*  */,
      0x00 /*  */
   };
   return T_ARRAY<T_TDC_INT_U8>(array_elements);
} // field_array_GMM007_0;
```

Message Sequence Chart of the primitive flow during execution of the generated test case by TAP:


MMI
GMM
MM
SIM
SIM_GMM_INSERT_IND-(0x801B0...
GMMREG_PLMN_MODE_REQ-(0x3304)
MMGMM_PLMN_MODE_REQ-(0x2E04)
GMMREG_ATTACH_REQ-(0x3300) (MS cla...
MMGMM_REG_REQ-(0x2E00)
MMGMM_REG_CNF-(0x6E00)
MMGMM_TMSI_IND-(0x6E0F) (TMSI c2)
GMMREG_PLMN_MODE_REQ-(0x3304)
MMGMM_PLMN_MODE_REQ-(0x2E04)

Primitives sent during test execution:



T	Nr	Time	Snd	Name	Rcv	Content
	1	00002002 ms	TAP	SIM_GMM_INSERT_IND	GMM	00 00 00 00 08 29 26 10 09 93 30 72 01 00
	2	00002002 ms	TAP	GMMREG_PLMN_MODE_REQ	GMM	(16*00) 54 41 50 00 00 00 00 00 47 4D 4D
	3	00017100 ms	GMM	MMGMM_PLMN_MODE_REQ	MM	(16*00) 54 41 50 00 00 00 00 00 47 4D 4D
	4	00003084 ms	TAP	GMMREG_ATTACH_REQ	GMM	04 02 01 00 E0 00 00 00 40 (15*00) 54 41
	5	00017450 ms	GMM	MMGMM_REG_REQ	MM	01 00 04 00
	6	00003454 ms	TAP	MMGMM_REG_CNF	GMM	01 02 06 02 00 01 0F 00 06 00 16 (17*00)
	7	00003474 ms	TAP	MMGMM_TMSI_IND	GMM	C2 (15*00) 54 41 50 00 00 00 00 00 47 4D
	8	00017800 ms	GMM	GMMREG_ATTACH_CNF	MMI	02 00 00 00 01 02 06 02 00 01 0F 00 06 00
	9	00003734 ms	TAP	GMMREG_PLMN_MODE_REQ	GMM	(16*00) 54 41 50 00 00 00 00 00 47 4D 4D
	10	00018100 ms	GMM	MMGMM_PLMN_MODE_REQ	MM	(16*00) 54 41 50 00 00 00 00 00 47 4D 4D

Element	Value	Cleartext
SIM_GMM_INSERT_IND	OPC: 0x801B0005	
op_mode (operation mod...	00	normal SIM card
imsi_field (international m...	08 29 26 10 09 93 30 72 01 00 00 00	<Sub structure>
c_field (length of imsi)	08	
field (imsi)	08 29 26 10 09 93 30 72 01	<Array>

A.2 Selected Parts of the Source Code

This final appendix section contains parts of the source code of the applications PCO and TCGen discussed in Subchapters 5.4 and 5.5, as well as of the frameworks explained in Subchapter 5.3. It is meant as a reference for the description of the implementational details and should solve eventually remaining problems in understanding the software concept.

The whole sources of the tools developed by the author consist of approximately 13000 lines of code (PCO: 10000, TCGen: 3000, without comments and empty lines).

A.2.1 Frameworks

Source Code 1 – Part of CMS interface (mentioned in 5.3):

```
/*************   PROCESS part   **************************************/
CMS_HANDLE p_create(          /* @func Create a CMS process (= thread)  */
   char * name,               /* @parm process name                     */
   CMS_ENTRY entry,           /* @parm entry function (start adress)    */
   int prio,                  /* @parm initial process priority         */
   long val,                  /* @parm parameter of entry()             */
   int stacksize );           /* @parm stack size in bytes              */
                              /* @returnvalue proc handle or error code */
CMS_HANDLE p_open(            /* @func Open a CMS process               */
   char * name );             /* @parm process name                     */
                              /* @returnvalue proc handle or error code */
/*************   SEMAPHORE part   ************************************/
CMS_RETURN r_request(         /* @func Request a CMS semaphore          */
   CMS_HANDLE rhandle,        /* @parm semaphore handle                 */
   CMS_TIME timeout );        /* time out time in msec                  */
                              /* @returnvalue CMS_OK or error code      */
CMS_RETURN r_release(         /* @func Release a CMS semaphore          */
   CMS_HANDLE rhandle );      /* @parm semaphore handle                 */
                              /* @returnvalue CMS_OK or error code      */
/*************   QUEUE part   ****************************************/
CMS_HANDLE q_create(          /* @func Create a CMS queue               */
   char * name,               /* @parm queue name                       */
   int msgnum,                /* @parm max # of messages                */
   int msgsize );             /* @parm max size of any message          */
                              /* @returnvalue queue handle or 0 | err   */
int q_write(                  /* @func Write a message into a queue     */
   CMS_HANDLE qhandle,        /* @parm queue handle                     */
   char * buf,                /* @parm message buffer                   */
   int msglen,                /* @parm message length                   */
   CMS_TIME timeout );        /* @parm time out time in msec            */
                              /* @returnvalue msg length or error code  */
...
```

Source Code 2 – Part of CCDATA interface (mentioned in 5.3):

```
/*
 * Selects the primitive for edit processing. The code of the
 * primitive (primcode) must be passed to the function.
 * The function returns a State and the primname in (*name)
 * and a handle for this edit process (phandle).
 * After a successful call the component maxCSize of the
 * phandle contains the size of the C-Structure for this primitive.
 */
extern USHORT CCDDATA_PREF(cde_prim_first)  (T_CCDE_HANDLE     * phandle,
                                             ULONG               primcode,
                                             char              * name);
/*
 * Get the next element of the selected primitive. All informations
 * of this element is stored into the element descriptor (pdescr).
 */
extern USHORT CCDDATA_PREF(cde_prim_next)   (T_CCDE_HANDLE     * phandle,
                                             UBYTE               descent,
                                             T_CCDE_ELEM_DESCR * pdescr);

/*
 * Selects at COMPOSITION (structure) for edit processing.
 * The short name (compname) of this composition must be passed
 * to this function.
 * The function returns a State and ahandle for this
 * edit process (chandle).
 * This function may be used for sub-structures (compositions)
 * of primitives and messages.
 * After a successful call the component maxCSize of the
 * chandle contains the size of the C-Structure for this composition.
 */
extern USHORT CCDDATA_PREF(cde_comp_first)  (T_CCDE_HANDLE     * chandle,
                                             T_ELM_SRC           source,
                                             char              * compname);

/*
 * Get the next element of the selected composition. All informations
 * of this element is stored into the element descriptor (cdescr).
 */
extern USHORT CCDDATA_PREF(cde_comp_next)   (T_CCDE_HANDLE     * chandle,
                                             UBYTE               descent,
                                             T_CCDE_ELEM_DESCR * descr);

/*
 * Read the value of the element out of the C-Structure (cstruct)
 * which containes a primitive or a decoded message. The infomations
 * of the element is stored in the input parameter edescr, wich is
 * previously assigned by a cde_xxxx_next () call. The value of this
 * element is stored in the memory area addressed by (*value).
 * After this call, the component symbolicValue of the edescr-struct
 * is updated with a symbolic value string, (if any defined).
 */
extern USHORT CCDDATA_PREF(cde_read_elem)   (T_CCDE_HANDLE     * handle,
                                             void              * cstruct,
                                             T_CCDE_ELEM_DESCR * edescr,
                                             UBYTE             * value);
```

Source Code 3 – Matrix of Entity Graph (mentioned in 5.3):

```
/* adjacence matrix. 1 = directly connected, 0 = not
   here symmetric = undirected graph,
   could later  be changed to a directed graph */

static char ad[MAXNODE][MAXNODE] =
{        /* M S S C S S M G R G D P L T R F L S P U P L R W U I L
            M I M C M S M M R R L L 2 3 L A L N P A K C R A D P 1
            I M S         M   R     R O P D C D P R T   L P P
                                                  T     P
                                                                  */
/* MMI  */  1,1,1,1,1,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,1,0,0,1,1,1,1,
/* SIM  */  1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
/* SMS  */  1,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,
/* CC   */  1,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
/* SM   */  1,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,
/* SS   */  1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
/* MM   */  1,1,1,1,0,1,1,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,
/* GMM  */  1,1,1,0,1,0,1,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,
/* RR   */  0,0,0,0,0,0,1,0,1,1,1,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,
/* GRR  */  0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,
/* DL   */  0,0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
/* PL   */  1,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
/* L2R  */  1,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,
/* T30  */  1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,
/* RLP  */  0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,1,
/* FAD  */  0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,
/* LLC  */  0,0,1,0,0,0,0,1,0,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,
/* SND  */  1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,0,0,0,0,0,0,
/* PPP  */  1,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,1,1,0,0,0,0,0,1,0,
/* UART */  1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,
/* PKT  */  1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,
/* LC   */  0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,
/* RRLP */  0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,
/* WAP  */  1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,
/* UDP  */  1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,
/* IP   */  1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,
/* L1   */  1,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,1
};
```

Source Code 4 – Entity Graph access functions (mentioned in 5.3):

```
/* ccddata_eg.c */
#ifndef CCDDATA_EG_C
CCDDATA_IMPORT_FUNC int    ccddata_eg_nodes (void);
CCDDATA_IMPORT_FUNC char** ccddata_eg_nodenames (void);
CCDDATA_IMPORT_FUNC char*  ccddata_eg_adjacent (int idx);
#endif /* !CCDDATA_EG_C */
```

Source Code 5 – TST reception interface (mentioned in 5.4.2):

```
extern int tst_register (const char* who);
extern int tst_withdraw (const char* who);
extern int tst_send (const char* string, const char* receiver);
```

A.2.2 PCO (written by the author)

Source Code 6 – Default Version of the PCO ini-file (mentioned in 5.4):

```
[General]
QueueSize=1400
ServerName=PCOS
DataSize=1600

[Viewers]
Str2IndPath=.

[Server]
TopMost=0
Tray=1
TestSessionPath=
DataQueueSize=100
SetTime=0
TimeStampPeriod=10

[Controller]
Primlist=0100010000
PrimFile=View.txt
StartList=pco_start.lst
StackConfigFile=pco_stack.xml
TSTCommunication=socket
TSTHeader=new
```

Source Code 7 – Creation of the Communication Header (mentioned in 5.4.1):

```
/******************************************************************/
U16 ipc_createMsg(      /* @func Create a message.               */
   void       *pvBuffer,/* @parm Buffer for the message to store. */
   U16         uwSize,  /* @parm Size of the buffer.              */
   MSG_HEADER  Msg      /* @parm Settings for the message to create.
  )                     /* @returnvalue One of the values below. */
…
{
CMS_PCB * pPCB;
CMS_RETURN rc;
int hChannel;
char acSnd[IPC_MAX_PATH_SIZE+NAMELEN+3];  /* abs addr of sender   */
char acRcv[IPC_MAX_PATH_SIZE+2];          /* abs addr of receiver */
MESSAGE * pHdr;
U8      * pubBuffer;
U16       uwHdr;
U16       uwSnd;
U16       uwRec;
U16       uwDat;
U16       uwHeader;

   /* (MK) buffer pointer must be valid                           */
   assert (pvBuffer);
   /* (MK) the buffer pointer must be aligned                     */
   assert (((U32)pvBuffer & (ALIGNMENT-1)) == 0);
   /* determining of absolute Sender-Adress: */
   if( Msg.pcSender == NULL )          /* NULL -> eig.Prozess-Name  */
   {
      rc = p_getcb( p_pid(), &pPCB );
      if( rc != CMS_OK )
         return 0;                     /* not a CMS process         */

      Msg.pcSender = pPCB->name;
   }
   if( *(Msg.pcSender) != '/' )
   {
      ipc_absAddr( acSnd, Msg.pcSender );
      Msg.pcSender = acSnd;
   }

   /* determining of absolute Receiver-Adress: */
   hChannel = (int)(Msg.pcReceiver); /* may be !!                  */
   if( (hChannel > 0) && (hChannel <= CMS_MAXPROC) &&
       (aChannel[hChannel].szSender != NULL) )
   {  /* IPC_HANDLE : abs. Addr. already known: */
      Msg.pcReceiver = aChannel[hChannel].szReceiver;
   }
   else
   {
      ipc_absAddr( acRcv, Msg.pcReceiver );
      Msg.pcReceiver = acRcv;
   }

   uwHdr = sizeof(MESSAGE);
   uwSnd = strlen (Msg.pcSender  ) + 1;
   uwRec = strlen (Msg.pcReceiver) + 1;
   uwDat = Msg.uwSize;

   uwHeader = uwHdr + uwSnd + uwRec;
   /* round up to align-ment */
   uwHeader = (uwHeader + (ALIGNMENT-1)) & ~(ALIGNMENT-1);

   if( (uwHeader + uwDat) > uwSize ) return 0;

   assert( uwHeader <  256 );

   pHdr      = (MESSAGE*)pvBuffer;
   pubBuffer = (unsigned char*)pvBuffer;

   ipc_getTime( &(pHdr->ulTime), &(pHdr->uwTenthOfMS) );

   pHdr->ubSize      = (U8) uwHeader;
   pHdr->uwSize      = uwDat;
   pHdr->uwID        = Msg.uwID;

   strcpy ((char*)&pubBuffer [uwHdr]          , Msg.pcReceiver);
   strcpy ((char*)&pubBuffer [uwHdr + uwRec]  , Msg.pcSender);
   if( uwDat )
      memcpy (&pubBuffer [uwHeader]    , Msg.pvBuffer, uwDat);
   return ((U16) pHdr->ubSize + pHdr->uwSize);
}
```

Source Code 8 – Registration of the “PCO”-queue (mentioned in 5.4.2):

```
PCOSrv_frameSupp::PCOSrv_frameSupp(const char* appl_path, int&err) :
  PCOSrv_core(appl_path, err)
{
  if (err!=0) return;

  /* wait for PCO-queue to be ready */
  CMS_HANDLE h;
  while ((h=q_open("PCO"))==CMS_NOTFND) {
    p_sleep(10);
  }
  q_close(h);

  /* register in FRAME context */
  tst_register("PCO");
}
```

Source Code 9 – Base classes for individual PCO viewers (mentioned in 5.4.3):

```
class PCOView_templ
{
public:
  PCOView_templ(const char* primq_name="", const char* ctrlq_name="");
  virtual ~PCOView_templ();

  int connect(void);
  int subscribe(const char* mobileId);
  virtual int on_connected(const void *buf,const char* sender);

  int disconnect(void);
  int unsubscribe(void);

  int open_logfile(const char* fname);
  int set_filter(const char* list, char prim_trace=0);

  int send2srv(void* buf, U16 size, U16 id);

  const char *primq_name() { return m_primq_name;}
  const char *ctrlq_name() { return m_ctrlq_name;}

  void set_srv_name(const char* sname);
  U8    srv_type() const { return m_srv_type;}

protected:
  char  m_primq_name[MAX_QNAME_LEN+1],m_ctrlq_name[MAX_QNAME_LEN+1];
  char  m_srvq_name[MAX_QNAME_LEN+1];
  U8    m_srv_type;
};
```

```
class PCOView_core : public PCOView_templ
{
public:
  PCOView_core(const char* ini_file, int& err,
      const char* primq_name="", const char* ctrlq_name="");
  virtual ~PCOView_core();

  virtual int dispatch_message(void* buf, U16 size, U16 id, const char* sender);

  virtual int interpret_message(void* buffer, U16 bufsize,
    void* &data, U16 &size, ULONG &id, U32 &time,
    char* &sender, char* &receiver) = 0;

  virtual void on_data(void* data, U16 size, ULONG id,
    U32 time, const char* sender, char* receiver) = 0;

  int propagate_inichange();
  virtual void on_inichange();

  bool  running() const {return m_running;}
  int dsize() const { return m_dsize;}

protected:
  CMS_HANDLE  m_prim_handle,m_ctrl_handle;
  bool  m_running;

  IniFile        *m_inifile;
  int             m_dsize;

  static  void cms_prim_proc(long view);
  static  void cms_ctrl_proc(long view);
};

class PCOView_frameSupp : public PCOView_core
{
public:
  PCOView_frameSupp(const char* ini_file, int& err,
                    const char* primq_name="",
                    const char* ctrlq_name="");
  virtual ~PCOView_frameSupp();

  virtual int interpret_message(void* buffer, U16 bufsize,
    void* &data, U16 &size, ULONG &id, U32 &time, char* &sender, char* &receiver);

  virtual int on_connected(const void *buf,const char* sender);

  int decode_tracestring(const char* instr, char* outstr, U16 size, ULONG &lastOPC);

  int check_version();
  int check_communication();

  virtual void on_inichange();
protected:
  char    m_tracestrBuf[DATA_MSG_MAX_SIZE];
};
```

A.2.3 TCGen (written by the author)

Source Code 10 – Default Version of the TCGen ini-file (mentioned in 5.5.2):

```
# default ini file for TCGenerator

# TDC include dir (e.g. \g23m\condat\ms\tdcinc)
tdcincdir   \g23m\condat\ms\tdcinc

# TDC include library (e.g. \g23m\condat\ms\tdclib\tdcinc.lib)
tdcinclib   \g23m\condat\ms\tdclib\tdcinc.lib

# TDC library (e.g. \GPF\LIB\win32\tdc.lib)
tdclib  \GPF\LIB\win32\tdc.lib

# Use PCON (0|1)
pcon    0

# Generate primitives with parameters (0|1)
parameters  1

# Rules file
rules   tcgen_rules.xml
```

Source Code 11 – Declaration of TCGen core class (mentioned in 5.5.1):

```
class TCGenerator_core : public PCOView_frameSupp
{
public:
  TCGenerator_core(const char* ini_file, int& err, const char* primq_name="TCGEN_VIEW",
                   const char* ctrlq_name="cTCGEN_VIEW");
  virtual ~TCGenerator_core();

  int generate(const char* pco_fname, const char* out_dir, char* entity_list,
               const char* project, const char* tcname);
  int analyze(const char* pco_fname);

  virtual int dispatch_message(void* buf, U16 size, U16 id, const char* sender);
  virtual int on_connected(const void *buf,const char* sender);
  virtual void on_data(void* data, U16 size, ULONG id, U32 time, const char* sender,
                       char* receiver);

  void set_rules(Rules *newrules);
  void set_tdcincdir(const char* tdcincdir) { strncpy(m_tdcincdir,tdcincdir,MAX_PATH_LEN);}
  void set_tdcinclib(const char* tdcinclib) { strncpy(m_tdcinclib,tdcinclib,MAX_PATH_LEN);}
  void set_tdclib(const char* tdclib) { strncpy(m_tdclib,tdclib,MAX_PATH_LEN);}
  void set_parameters(bool parameters) { m_parameters=parameters;}
protected:
  int          m_pass_nr;
  Rules*       m_rules;
  ...
  int write_params(Next_func func, T_CCDE_HANDLE &h,
                   const char* data,
                   int level, const char *union_name="");
  int write_initials();
  void finalize_initials();
};
```

Source Code 12 – Main function of TCGen (mentioned in 5.5.1):

```
int main(int argc, const char* const argv[])
{
  /* read cmdline, source TCGen ini-file and initialize m_rules from xml-file */
  if (read_cmdline(argc, argv)==-1) return 0;

  TCGenerator_cmdl generator(inif, ret);
  if (ret!=0)
  {
      /* error handling */
  }
  else
  {
    generator.set_parameters(m_parameters);
    if (m_analyse)
    {
      ret=generator.analyze(m_pco_file);
      if (ret<0)
      {
        /* error handling */
      }
      else
      {
        printf("analyzing test session ...\n");
        m_running=1;
      }
    }
    else
    {
      generator.set_rules(&m_rules);
      generator.set_tdcincdir(m_tdcincdir);
      generator.set_tdcinclib(m_tdcinclib);
      generator.set_tdclib(m_tdclib);
      ret=generator.generate(m_pco_file,m_out_dir,m_entity_list,m_project,m_tcname);
      if (ret<0)
      {
        /* error handling */
      }
      else
      {
        printf("generating testcase ...\n");
        m_running=1;
      }
    }
    while (generator.running() && m_running)
    {
      p_schedule();
      p_sleep((CMS_TIME)100);
    }
  }
  write_inifile();
  m_rules.store_as_XML();
  return ret;
}
```

Source Code 13 – Starter function for generation (mentioned in 5.5.1):

```
int TCGenerator_core::generate(const char* pco_fname, const char* out_dir,
                               char* entity_list,
                               const char* project, const char* tcname)
{
  ...
  strcpy(m_outdir,out_dir); _mkdir(m_outdir);
  strcpy(m_pco_fname,pco_fname);
  strcpy(m_out_dir,out_dir);

  memset(m_entities_ut,0,sizeof(m_entities_ut));
  /* extract comma separated entity names */
  ...

  if (strlen(project))
  {
    strcpy(m_projname,project);
  }
  else
  {
    strcpy(m_projname,m_entities_ut[0]);
  }

  /* build long name of testcase */
  ...

  // prepare generation
  m_pass_nr=-1;
  m_prim_nr=0;
  m_array_nr=m_array_prenr=0;
  m_struct_nr=m_struct_prenr=0;
  m_sdu_nr=m_sdu_prenr=0;
  m_last_time=0;
  m_new_elem_found=true;
  m_new_prim_found=false;
  m_header_creation=false;
  memset(m_entities,0,sizeof(m_entities));
  memset(m_saps,0,sizeof(m_saps));
  memset(m_aims,0,sizeof(m_aims));
  memset(m_prims,0,sizeof(m_prims));
  m_mode=TCGEN_MODE_GEN;
  m_prim_count=0;
  m_noint_prim_count=0;

  // request logfile for the first time
  if (open_logfile(pco_fname)<0)
  {
    m_mode=TCGEN_MODE_IDLE;
    return -3;
  }

  return 0;
}
```

Source Code 14 – Handling of basic type elements (mentioned in 5.5.2):

```
...
case T_byte: case T_short: case T_long:
{
  if (level!=m_pass_nr-1)
  {
    return 1;
  }

  char *type="";
  switch (desc.btype)
  {
    case T_byte:
    {
      type="T_TDC_INT_U8";
      sprintf(val_str,"0x%02X",*(U8*)value);
      break;
    }
    ...
  }

  if (!m_parameters || level>0)
  {
    // try to apply rule
    if (m_rules && m_rules->change_param(desc.sname,val_str,changed_val_str,MAX_VALSTR_LEN)==TCGEN_RULES_OK)
    {
      val=changed_val_str;
      sprintf(comment_str,"<change rule applied on %s>",val_str);
      comment=comment_str;
    }
    else
    {
      val=val_str;
      comment=desc.symbolicValue;
    }

    if (aelem) // array element
    {
      fprintf(m_constraints_cppfile,"      %s /* %s */",val_str,desc.symbolicValue);
    }
    else
    {
      fprintf(m_constraints_cppfile,"   %s%s->%s%s%s= %s; // %s\n",skip,context,union_name,ptr,desc.sname,val,comment);
    }
  }
  else
  {
    if (m_header_creation)
    {
      fprintf(m_constraints_cppfile,"%s %s",type,desc.sname);
      fprintf(m_constraints_hfile,"%s %s",type,desc.sname);
    }
    else
    {
      fprintf(m_steps_cppfile,"%s /* %s */",val_str,desc.symbolicValue);

      if (m_new_prim_found)
      {
        // try to apply rule
        sprintf(val_str,"%s",desc.sname);
        if (m_rules && m_rules->change_param(desc.sname,val_str,changed_val_str,MAX_VALSTR_LEN)==TCGEN_RULES_OK)
        {
          val=changed_val_str;
          sprintf(comment_str,"<change rule applied on %s>",val_str);
          comment=comment_str;
        }
        else
        {
          val=val_str;
          comment="";
        }

        fprintf(m_constraints_cppfile,"   %s%s->%s%s%s= %s; // %s\n",skip,context,union_name,ptr,desc.sname,val,comment);
      }
    }
  }
```

Source Code 15 – Handling of structure elements (mentioned in 5.5.2):

```
case T_struct:
{
  if (level==m_pass_nr-1)
  {
    if (!m_parameters || level>0)
    {
      fprintf(m_constraints_hfile,"extern T_%s %s_%s_%u();\n",desc.aname,desc.sname,m_tcname,m_struct_prenr);

      // try to apply rule
      sprintf(val_str,"%s_%s_%u()",desc.sname,m_tcname,m_struct_prenr);
      if (m_rules && m_rules->change_param(desc.sname,val_str,changed_val_str,MAX_VALSTR_LEN)==TCGEN_RULES_OK)
      {
        val=changed_val_str;
        sprintf(comment_str,"<change rule applied on %s>",val_str);
        comment=comment_str;
      }
      else
      {
        val=val_str;
        comment="";
      }
      if (aelem)
      {
        fprintf(m_constraints_cppfile,"     %s /* %s */",val_str,desc.symbolicValue);
      }
      else
      {
        fprintf(m_constraints_cppfile,"  %s%s->%s%s%s= %s; // %s\n",skip,context,union_name,ptr,desc.sname,val,comment);
      }
      m_struct_prenr++;
      return 1;
    }
    else
    {
      if (m_header_creation)
      {
        fprintf(m_constraints_cppfile,"T_%s %s",desc.aname,desc.sname);
        fprintf(m_constraints_hfile,"T_%s %s",desc.aname,desc.sname);
      }
      else
      {
        if (m_new_prim_found)
        {
          // try to apply rule
          sprintf(val_str,"%s",desc.sname);
          if (m_rules && m_rules->change_param(desc.sname,val_str,changed_val_str,MAX_VALSTR_LEN)==TCGEN_RULES_OK)
          {
            val=changed_val_str;
            sprintf(comment_str,"<change rule applied on %s>",val_str);
            comment=comment_str;
          }
          else
          {
            val=val_str;
            comment="";
          }

          fprintf(m_constraints_cppfile,"  %s%s->%s%s%s= %s; // %s\n",skip,context,union_name,ptr,desc.sname,val,comment);
        }

        fprintf(m_constraints_hfile,"extern T_%s %s_%s_%u();\n",desc.aname,desc.sname,m_tcname,m_struct_prenr);
        fprintf(m_steps_cppfile,"%s_%s_%u()",desc.sname,m_tcname,m_struct_prenr);
        m_struct_prenr++;
      }
      return 1;
    }
  }
  if (level==m_pass_nr-2)
  {
    fprintf(m_constraints_cppfile,"T_%s %s_%s_%u()\n",desc.aname,desc.sname,m_tcname,m_struct_nr);
    fprintf(m_constraints_cppfile,"{\n");
    fprintf(m_constraints_cppfile,"   T_%s pstruct;\n",desc.aname);
  }
  T_CCDE_HANDLE struct_handle;
  if (cde_comp_first (&struct_handle,desc.esource,desc.aname) == CCDEDIT_OK) {
    while (write_params((Next_func)cde_comp_next,struct_handle,
      data+desc.offset,level+1)) {}
  }
  if (level==m_pass_nr-2)
  {
    fprintf(m_constraints_cppfile,"   return pstruct;\n");
    fprintf(m_constraints_cppfile,"} // %s_%s_%u\n\n", desc.sname,m_tcname,m_struct_nr);
    m_struct_nr++;
  }
```

Source Code 16 – Declaration of class `Rules` (mentioned in 5.5.2):

```
struct SkipParam
{
  string  m_name;
  SkipParam() {}
  SkipParam(const string& name) :  m_name(name) {}
};
typedef list<SkipParam> SkipParamList;

struct SkipPrim
{
  string        m_name;
  SkipParamList m_params;
  bool          m_full;
  SkipPrim() {}
  SkipPrim(const string& name, bool full=false) :  m_name(name), m_full(full) {}
};
typedef list<SkipPrim> SkipPrimList;

struct ChangeParam
{
  string  m_name;
  string  m_change_statement;
  ChangeParam() {}
  ChangeParam(const string& name, const string& change_statement) :
    m_name(name), m_change_statement(change_statement) {}
};
typedef list<ChangeParam> ChangeParamList;

struct ChangePrim
{
  string          m_name;
  ChangeParamList m_params;
  string          m_change_statement;
  ChangePrim() {}
  ChangePrim(const string& name) :  m_name(name) {}
};
typedef list<ChangePrim> ChangePrimList;

class Rules : public XmlNotify
{
public:
  int store_as_XML(const char* fname=NULL);
  int load_from_XML(const char* fname);

  Rules();
  virtual ~Rules();

  void add_skip(const char* prim, const char* param=NULL);
  void add_change(const char* prim,const char* param,const char* change_statement);

  bool   skip_prim(const char* prim);
  void   set_curr_prim(const char* prim);
  bool   skip_param(const char* param);
  int    change_prim(const char* prim, char* new_val, unsigned int new_val_size);
  int    change_param(const char* param, const char* val, char* new_val,
                 unsigned int new_val_size);

  virtual void foundNode    ( string & name, string & attributes );
  virtual void foundElement ( string & name, string & value, string & attributes );
  virtual void startElement ( string & name, string & value, string & attributes );
  virtual void endElement   ( string & name, string & value, string & attributes );

  U32   max_timegap() const { return m_max_timegap;}
  bool  timeouts() const { return m_timeouts;}

protected:
  U32   m_max_timegap;
  bool  m_timeouts;
  string  m_fname;

  SkipPrimList    m_skip_prims;
  ChangePrimList  m_change_prims;

  SkipPrimList    m_current_skip_prims;
  ChangePrimList  m_current_change_prims;


  string getAttr(const string& attributes, string aname) const;
  bool compare_name(const char* name, const char* wildcard_name) const;
  int  change_val(const char* val, const char* change_statement, char* new_val,
               unsigned int new_val_size) const;
  void clean();
};
```

Source Code 17 – Example of a rules file (mentioned in 5.5.2):

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Rules for TCGenerator -->
<tcgen>
   <options max_timegap="0" timeouts="0"></options>
   <skip primitive="GMMRR_EXAMPLE_*"></skip>
   <skip primitive="LLGMM_EXAMPLE_REQ"></skip>
   <skip primitive="sim_gmm_example_req" param="kc"></skip>
   <change primitive="MMGMM_EXAMPLE_REQ" param="detach_cause">
      (%v)=1 ? 0x5 : %v)
   </change>
   <change primitive="mmgmm_example_acc_req" param="mnc">empty_array</change>
   <change primitive="mmgmm_example_*">TEST_%v</change>
</tcgen>
```

Source Code 18 – Changing of values in class `Rules` (mentioned in 5.5.2):

```
int  Rules::change_val(const char* val, const char* change_statement,
                       char* new_val, unsigned int new_val_size) const
{
  unsigned int q=0, pos=0;
  while (change_statement[q])
  {
    if (pos>=new_val_size)
    {
      // buffer to small
      return TCGEN_RULES_ERR_BUFTOSMALL;
    }
    if (change_statement[q]=='%')
    {
      q++;
      if (change_statement[q]=='v' || change_statement[q]=='V')
      {
        if (pos+strlen(val)>=new_val_size)
        {
          // buffer to small
          return TCGEN_RULES_ERR_BUFTOSMALL;
        }
        // insert original value
        new_val[pos]=0;
        strcat(new_val,val);
        q++; pos+=strlen(val);
        continue;
      }
    }
    // just copy character
    new_val[pos]=change_statement[q];
    q++; pos++;
  }
  // finalize new value
  new_val[pos]=0;
  return TCGEN_RULES_OK;
}
```

List of Terms and Abbreviations

ACI	Application Control Interface (AT Command Interface)
ADA	Programming language designed in 1979 and named after Augusta Ada Byron
Air message	Block of data, message, sent via the cellular network; can be contained in a primitive
Alpha-Test	Test done by the developer himself
AT commands	AT is originally a contraction of attention; AT commands where firstly used to program SmartModems from HAYES MICROCOMPUTER PRODUCTS; they are just ASCII-strings sent to/from a device
ASCII	American Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
Beta-Test	Test done by persons not involved in the development
Black-Box	System of which only the outer interface is known
BNF	Backus-Naur Form
Bottom-Up	Starting with the smallest parts of a system
CCD	Condat Coder and Decoder
Clipboard	Area of temporary memory used to transfer text or graphics
Code Completion	Feature supported by various development environments, providing quick information about the context of, e.g., structures or functions the user points on
Code Walkthrough	Manually examination of source code (see 2.3)
Compiler	A computer program that translates a high-level programming language into machine language
CPU	Central Processing Unit
Data-driven	Processes influenced by the data handled by specific software
Debugging	Attempting to determine the cause of the symptoms of software malfunctions detected by testing or by frenzied user complaints

DLL	Dynamic Linked Library
Domain	Generally a limited region or field marked by some specific property
Entity	OSI terminology for a layer protocol machine; An entity within a layer performs the functions of the layer within a single computer system, accessing the layer entity below and providing services to the layer entity above at local service access points (SAPs)
ETSI	European Telecommunications Standards Institute
EUT	Entity Under Test
Field Test	Test done anywhere in the country and under real environmental conditions (see 3.2)
FRAME	Runtime environment of the protocol stack, product of TEXAS INSTRUMENTS Berlin
FTA	Final Test Approval (see 3.2)
Function-driven	Processes influenced by the functions implemented by specific software
GMM	GPRS Mobile Management
GNU	GNU’s Not Unix
GPRS	General Packet Radio Service
Graph	An object consisting of vertices (or nodes) and edges (or arcs) between pairs of vertices
Grey-Box	Black-/ White-Box mixture
GRR	GPRS Radio Resource
GSM	Global Standard for Mobile Communication
GSMS	GPRS Short Message System
GUI	Graphical User Interface
Hardware	The physical part of a computer system; the machinery and equipment
HTML	Hyper Text Markup Language
IEEE	Institute of Electrical and Electronics Engineers, Inc; the world's largest technical professional society
Image-File	Binary file which can be loaded on a test board, containing instructions for the processor at the board, e.g., protocol stack software for a mobile phone

Instrumentation	Adding of special debugging code to a software
IOT	In-Orbit Test (see 3.2)
IP	Internet Protocol
ISO	International Standards Organization
ITU	International Telecommunications Union
Java	A cross-platform programming language from Sun Microsystems (see [JAVA], [JAVABOOK])
Linker	A program that combines one or more files containing object code from separately compiled program modules into a single file containing loadable or executable code
Linux	Operating system based on open sources (e.g., for PCs), Linux Is Not UniX
LL	Logical Link
LLC	Logical Link Control
Memory Supervision	Mechanism to control the memory access of software parts to detect leaks or access violations
MFC	Microsoft Foundation Classes, common C++ classes provided by MICROSOFT for easy access to the WINDOWS GUI (see [MFC])
MM	Mobile Management
MMI	Man Machine Interface
MNSMS	Mobile Network Service for Short Message System
MS	MICROSOFT; software company - see [MICROSOFT]
MSC	Message Sequence Chart
Nucleus	Real time operating system by ACCELERATED TECHNOLOGY (see [AT])
Object	In the sense of object-oriented programming languages it is a component containing both data and instructions for the operations to be performed on that data
Object-oriented	Having to do with or making use of objects
Official Approval	Test using test cases provided by the ETSI or network operators (see 3.2)
One-Source-Concept	Approach where all documents used in the different stages of a software project are derived from the speci-

	fication, e.g., source code and test cases
OO	Object-Oriented
OPC	OPerating Code
Operating Code	Unique ID identifying a specific primitive with its parameters
Oracle	Any means used to predict the outcome of a test
OSI	Open System Interconnection
Passive Task Model	Mechanism providing a general main function to tasks inside which, e.g., messages are received and delivered to individual callback functions
PC	Personal Computer
PCO	Point of Control and Observation
PCO-Controller	Controlling part of PCO (see 5.4.2)
PCO-Protocol	Set of control messages exchanged between components of PCO
PCO-Server	Fundamental part of PCO (see 5.4.2)
PCO-Viewer	Visualizing part of PCO (see 5.4.3)
PEI	Protocol stack Entity Interface (see 5.3)
PIN	Personal Identification Number
Primitive	Block of data which is exchanged between entities of a protocol stack
Process	The sequence of states of an executing
Processor	The part of a computer that interprets and carries out the instructions contained in the software; the CPU
Protocol Stack	A hierarchy of protocols which work together to provide the services on a communications network
PS	Protocol Stack
Pseudo Module	Module exporting a specified interface but containing no real functionality
QT	Multiplatform C++ application framework with GUI support
Quick Test	Short test usually applied after small software changes
Regression	Act of returning to a previous state
Robustness Test	Test checking the robustness of software over long

	time periods
Router	A device that finds the best path for a data packet to be sent from one network to another
RR	Radio Resource
RTOS	Real Time Operating System
Server	The computer in a client/server architecture that supplies files or services
SAP	Service Access Point; a data interface between two layers of a protocol stack
SAS	Stand Alone Simulator
SAT	Stand Alone Tester
Script	A series of instructions for a computer
SDL	Standard Definitions Language
Simulation Test	Test on a PC-simulation of the PS (see 3.2)
SMS	Short Message System
SNDCP	Sub-Network Dependent Convergence Protocol
Software	A set of instructions executed by a computer
Stack	Abbreviation of Protocol Stack
STL	Semantic Transfer Language (see [POSTON])
STL	Standard Templates Library for C++ (see [STL])
System Tray	Location on the far right of the WINDOWS taskbar
TAP	Test Application Process
TAP-Caller	GUI application for calling the TAP tool
Target-Test	Test on an actual hardware target (see 3.2)
TCGen	Test Case Generator
Tcl	Tool Command Language (see [TCL])
TDC	Test Description Code
Test, Testing	The process of exercising a product to identify differences between expected and actual behavior
Test Case	Description of a special test which can, e.g., be used by a dedicated tool to run this test
Tk	Graphical user interface toolkit that makes it possible to create powerful GUIs (see [TCL])

Thread	One part of a larger program that can be executed independent of the whole
TI	TEXAS INSTRUMENTS
Tool	Software program used primarily to create, manipulate, modify, or analyze other programs
Top-Down	Starting with a whole system and finally coming to the smallest parts of it
Trace	String containing specific information about the internal status of a part of software (e.g., an entity)
Trace-class	Group of traces with common meaning
Tracing	Visualization of information about the internal state of a running (mobile) software using traces and duplicated primitives
TST	TeST Interface
TTCN-3	Testing and Test Control Notation, version three
UDP	User Datagram Protocol
UML	Unified Modeling Language (see [UML])
UMTS	Universal Mobile Telecommunication System (see [UMTS])
UNIX	Multi-user operating system developed by AT&T's Bell Laboratories in the USA during the late 1960s
USB	Universal Serial Bus
VCMS	Virtual Condat Multitasking System (see 5.3)
Viewer-Interface	Set of functions and constraints an application has to support to connect to PCO-server (see 5.4.3)
VSI	Virtual System Interface (see 5.3)
VxWorks	Real time operating system by WIND RIVER SYSTEMS (see [WRS])
WAP	Wireless Application Protocol
White-Box	System of which all internal structures are known
Windows	PC operating system by MICROSOFT
XML	eXtensible Markup Language (see [XML])
xPanel	eXtended Panel, tool which emulates a virtual mobile (see [TRACING])

List of References

[AGILENT] ....http://www.agilent.com/; 10-21-2003

[ANITE] ....http://www.anite.com/; 10-21-2003

[APTEST]...............http://www.aptest.com/; 10-21-2003

[ARIANE] http://java.sun.com/people/jag/Ariane5.html; 10-21-2003

[ASN1] http://www.asn1.org/; 10-21-2003

[ASN1BOOK].........Larmouth, J.: “ASN.1 Complete”; Morgan Kaufmann Publishers; 1999

[AT] ....http://www.acceleratedtechnology.com/; 10-21-2003

[AUTOMATION] ...Fewster, M., Graham, D.: “Software Test Automation”; Addison-Wesley Professional; 2000

[BALZERT] ....Balzert, H.: “Lehrbuch der Software-Technik 1: Software-Entwicklung“; 2. edition, Spektrum-Verlag; Heidelberg, 2000

[BAUMG] ....Baumgarten, B.: “OSI Conformance Testing Methodology and Ttcn”; Elsevier Science Pub Co; 1994

[BEIZER] ....Beizer, B.: “Black Box Testing“; John Wiley & Sons Inc.; 1995

[BETA]....................Fine, M.R.: “Beta Testing for Better Software”; John Wiley & Sons; 2002

[CINDERELLA].....http://www.cinderella.dk/; 11-11-2003

[CPP] ....Eckel, B.: “Thinking in C++”; Prentice Hall; 2003

[CONDAT] ....http://www.condat.de; 10-21-2003

[DISASTERS].........http://www.ima.umn.edu/~arnold/disasters/; 10-21-2003

[DISASTERS2].......http://www-aix.gsi.de/~giese/swr/att1.html; 10-21-2003

[DISASTERS3].......http://www.year2000.com/bugbytes/NFbugbytes.html; 10-21-2003

[ESA].......................http://www.esa.int/; 10-21-2003

[ETSI]......................http://www.etsi.org/; 10-21-2003

[EVALG]................http://www.brics.dk/Activities/95/EvolvAlg/; 10-21-2003

[FORMAL].............Turner, J.G., McCluskey, T.L.: “The construction of formal specifications“; McGraw-Hill; London, 1994

[FRAME]................frame_users_guide.doc; TI internal documentation; Berlin, 2003

[FRAPPIER]............Frappier, M.: “Software Specification Methods: An Overview Using a Case Study (Formal Approaches to Computing and Information Technology)“; Springer Verlag; 2000

[GIMPEL] ....http://www.gimpel.com/; 11-03-2003

[GNU]......................http://www.gnu.org/; 10-21-2003

[GOODEN] ....Goodenough, J.B., Gerhart, S.L.: “Toward a Theory of Test Data Selection” in “Current Trends in Programming Methodology”, Vol. 2; R.T. Yeh (ed), Prentice-Hall; 1977

[GPRS] ....................T.O.P. BusinessInteractive: “GPRS Basics“; J.Schlembach Fachverlag; 2003

[GRAHAM].............Graham, D.R..: “Software Verification and Testing Tools: Availability and Uptake” in “Proceedings SE90”; Hall, P.A.V. (ed.); Brighton, 1990

[GRAHAM2]...........Graham, D.R: “Computer Aided Software Testing: The CAST Report”; Unicorn Seminars; U.K., 1990

[GRAPH].................Harris, J.M., Hirst, J.L., Mossinghoff, M.J.: “Combinatorics and Graph Theory”; Springer-Verlag; 2000

[GSM]......................T.O.P. BusinessInteractive: “GSM Basics“; J.Schlembach Fachverlag; 2003

[GTB] ....German Testing Board: http://www.asqf.de/deu/tester/board/index.php; 10-21-2003

[HARDWARE] ....Meyers, M.: “Introduction to PC Hardware and Troubleshooting“; Osborne McGraw-Hill; 2003

[HORN] ....Horn, E.: lecture slides “Grundlagen der Softwareentwicklung“; University of Potsdam; Potsdam, 1996

[HOWDEN].............Howden, W.E.: “A Functional Approach to Program Testing and Analysis”; IEEE Transactions on Software Engineering 12; 1986

[IEEE]......................http://www.ieee.org/; 11-10-2003

[IEEE91]..................IEEE Standard 829-1991: “Standard for Software Test Documentation”; IEEE Press; New York, 1991

[IIST].......................Institute for Software Testing: http://www.softdim.com/iist/; 10-21-2003

[INTEL]...................Intel Corporation: http://www.intel.com/; 10-21-2003

[ISOOSI] ....http://www.iso.org/; ISO Nr.: 35.100; 10-21-2003

[ITU]........................http://www.itu.int/; 11-10-2003

[JASC].....................http://www.jasc.com/; 10-21-2003

[JAVA]....................http://java.sun.com/; 10-21-2003

[JAVABOOK].........Eckel, B.: „Thinking in Java"; Prentice Hall; 2002

[JONES] ....Jones, C.: "Applied Software Measurements: Assuring Productivity and Quality"; McGraw-Hill; New York, 1991

[KANER].................Kaner, C., Falk, J., Nguyen, H.Q.: "Testing Computer Software", 2. edition; Van Nostrand Reinhold; 1993

[KIT]........................Kit, E.: "Software Testing in the Real World"; ACM Press; 1995

[MFC]......................Shepherd, G., Wingo, s.: "MFC Internals. Inside the Microsoft Foundation Class Architecture"; Addison-Wesley Professional; 1996

[MICROSOFT] ....http://www.microsoft.com; 10-21-2003

[MSC]......................ITU-T SG 10: "Message Sequence Chart (MSC)", Rec. Z.120; Geneva, 2000

[MYERS].................Myers, G.J.: "Methodisches Testen von Programmen", 5. edition; R. Oldenbourg Verlag GmbH; München, 1995

[NASA] ....http://www.nasa.gov; 10-21-2003

[OMAR89] ....Omar, A.A., Mohammad, F.A.: "Structural Testing of Programs" Softw Eng Notes, Vol 14, No 2; ACM Sigsoft; 1989

[OMAR91] ....Omar, A.A., Mohammad, F.A.: "A Survey of Software Functional Testing Methods", Softw Eng Notes, Vol 16, No 2; ACM Sigsoft; 1991

[PERRY] ....Perry, W.: "Effective Methods for Software Testing"; John Wiley & Sons Inc.; 1995

[POSTON]...............Poston, R.M.: "Automating Specification-Based Software Testing"; Institute of Electrical and Electronics Engineers; 1996

[REED]....................Reed, R., Reed, J.: "SDL 2001: Meeting UML"; Springer-Verlag; Heidelberg, 2001

[RSD]......................http://www.rsd.de/; 10-21-2003

[SAPE].....................Vogler, T.: "Datentyp- und Interfaceeditor für Mobilfunkprotokolle auf der Basis von XML"; diploma thesis at

“Fachhochschule Konstanz“; 2003

[SCHIRM]...............http://www.softwaretesting.de/article/view/45/1/7/; 10-21-2003

[SDL]........................Mitschele-Thiel, A.: “Systems Engineering with SDL”; John Wiley & Sons; 2001

[SIPSER] ....Sipser, M.: “Introduction to the Theory of Computation”; PWS Publishing Company ; 1997

[SOFTHARD] ....Petersen, D.A., Patterson, Indurkhya, N.: “Computer Organization and Design Second Edition: The Hardware/Software Interface”; Morgan Kaufmann Publishers; 1997

[SOFTWARE].........Cockburn, A.: “Agile Software Development”; Addison-Wesley Professional; 2001

[STL] ....Meyers, S.: “Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library“; Addison-Wesley Professional; 2001

[STOCKS]...............Stocks, P., Carrington D.: “Deriving Software Test Cases from Formal Specifications”; Proceedings Australian Software Engineering Conference; 1991

[TCL]........................http://www.tcl.tk/; 10-21-2003

[THERAC] ....Turner, C.S.: "An Investigation of the Therac-25 Accidents”;
http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html; 10-28-2003;
reprinted from “IEEE Computer”, Vol. 26, No. 7, pp. 18-41; July 1993

[TDC] ....8434_510_02_tdc_user_guide.doc; TI internal documentation; Berlin, 2003

[TELELOGIC] ....http://www.telelogic.com; 10-21-2003

[TESTTECH] ....http://www.testingtech.de; 12-09-2003

[TI]...........................http://www.ti.com; 10-21-2003

[TRACING].............Kießling, R.: „Tracing … an Essential Part of Software Testing in the Mobile Business”; undergraduate thesis at University of Potsdam; Königs Wusterhausen, 2003

[TRAUBOTH].........Trauboth, H.: “Software-Qualitätssicherung. Konstruktive und analytische Methoden”; 1993

[TROLL] ....http://www.trolltech.com/; 10-21-2003

[TTCN3]..................TTCN-3 standards; http://www.etsi.org/frameset/home.htm?/ptcc/ptccttcn3download.htm; 11-03-2003

[UML]......................Fowler, M.: ”UML Distilled. A Brief Guide to the Standard Object Modeling Languange”; Addison-Wesley Professional; 2003

[UMTS] ....Holma, H., Toskala, A.: “WCDMA for UMTS”; Jonh Wiley & Sons, Ltd; 2002

[VTEST]..................Arnold, T.R.: “Software Testing with Visual Test 4.0”; Hungry Minds, Inc; 1996

[WRS]......................http://www.windriver.com/; 10-21-2003

[XML]......................Harold, E.R., Means, W.S.: “XML in a Nutshell. A Desktop Quick Reference”; O'Reilly & Associates; 2002

Statement

I, Ronny Kießling, born on January 22th in 1976, hereby confirm that

- the diploma thesis titled with

  “Automated Generation of Software Test Cases for Mobile Business Based on Tracing Mechanisms”

  has been written by me, guided by Professor Dr. Jürgensen and Dr. Schmidt, and that I did not use any sources other than the ones listed at the end of the paper;

- I denoted explicit quotations and the usage of opinions of other authors throughout the document.

[illegible]

Ronny Kießling, Königs Wusterhausen, 2004-02-23

Deutsche Inhaltsangabe (German Abstract)

Thema:	Automatische Generierung von Software-Testfällen für den Mobilfunk-Bereich auf Basis von Tracing-Mechanismen
Author:	Ronny Kießling
Studiengang:	Diplom-Informatik
Schlüsselworte:	Mobilfunk; Testfall-Generierung; tracing; C++; XML

In der vorliegenden Diplomarbeit wird ein Generator für Software-Testfälle im Bereich Mobilfunk präsentiert, der, anders als viele bekannte Generator-Programme, nicht Software-Spezifikationen oder Implementierungen als Eingabe erwartet, sondern Daten, die während vorangegangener Testläufe aufgezeichnet wurden. Auf den ersten Blick mutet dieser Ansatz etwas seltsam an, aber verschiedene Anwendungsmöglichkeiten, z.B. für Regressions-Tests oder zur Reproduktion von fehlerhaftem Verhalten, konnten bereits identifiziert werden.

Die tatsächliche Implementierung einer ersten Programm-Version wurde vom Autor während seiner studentischen Tätigkeit bei der Firma TEXAS INSTRUMENTS BERLIN AG durchgeführt. Im Zusammenhang mit der theoretischen Konzeption sowie der Realisierung des Testfall-Generators werden das verwendete Software-Rahmenwerk und das ebenfalls vom Autor entwickelte Logging-Programm beschrieben. Die Diplomarbeit enthält eine Analyse der Anforderungen bezüglich Benutzer-Schnittstelle, Performance und Zuverlässigkeit und berichtet über Probleme und Lösungen während der Implementierung. Weiterhin erfolgt eine Untersuchung über Möglichkeiten, die Test-Programme selbst zu testen. Abschließend werden noch bestehende Probleme und Einschränkungen diskutiert und Pläne für zukünftige Weiterentwicklungen vorgestellt.