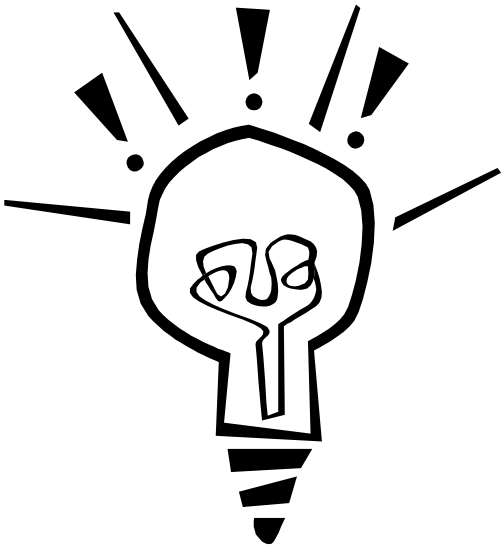
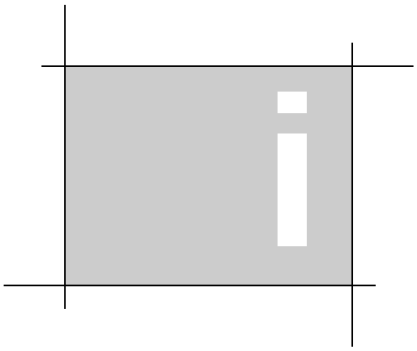
CODEN: LUTEDX(TETS-5375)/1-98/(1999) & local 35

Reverse engineering PLEX-C code to SDL10 code

Martin Berg
Dep. of Communication Systems
Lund Institute of Technology

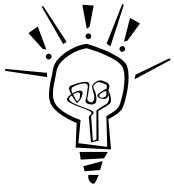
Tutors:
Magnus C. Ohlsson (Dep. of Communication Systems, LTH Lund)
Jörgen Palm (Ericsson Radio Systems AB, Hässleholm)
Henrik Cosmo (Ericsson Radio Systems AB, Hässleholm)





Abstract

The telecom business is one of the fastest growing markets today. Many companies are fighting over the market shares and to be at the top of it, the companies have to make their product developments more efficient and with higher quality. Increasing one of these components will probably decrease the other. This can be avoided by using the reverse engineering technology. It can be applied on a system to increase the quality, shorten the development time, or both. In this thesis we discuss different ways to increase quality and shorten the development time, by applying reverse engineering. One solution can be a software programming language change, and the new language may describe the system at a higher level of abstraction. It is that solution we have focused on in our work. Parts in Ericsson's GSM system is converted from their old programming language PLEX-C to the graphical programming language SDL10. The purpose is to develop features in the SDL10 environment in the future, which will both increase the quality and shorten the development time. Conversion between two programming languages is not an easy task. The differences between the two languages address some problems. These problems and solutions to them are discussed and presented in this thesis.



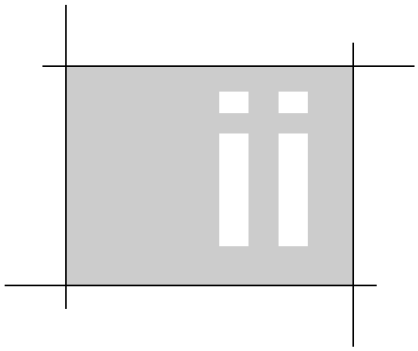
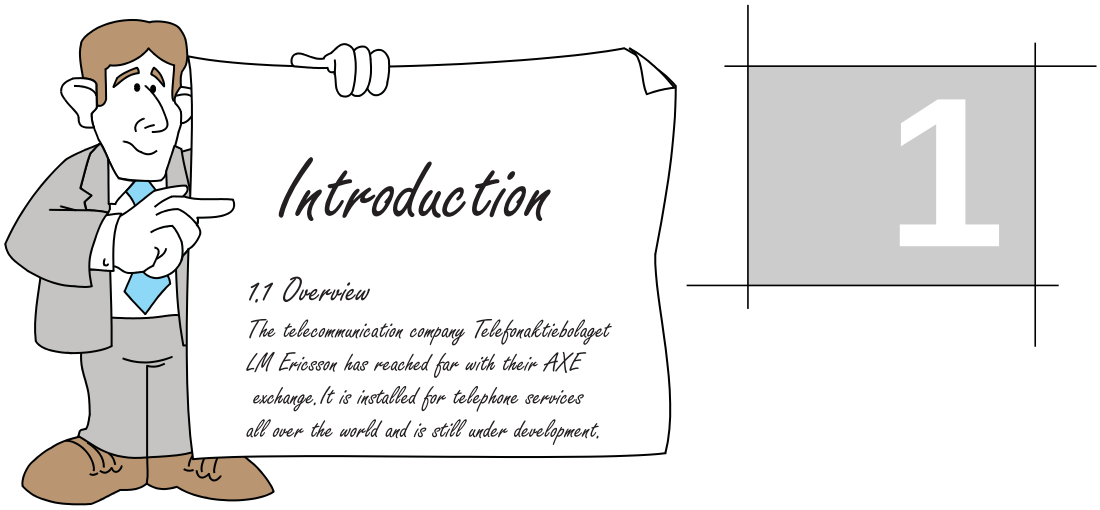


Table of Contents



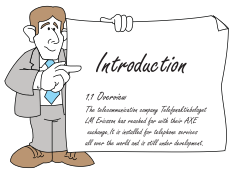
Introduction

1.1 Overview

The telecommunication company *Telefonaktiebolaget LM Ericsson* has reached far with their AXE exchange. It is installed for telephone services all over the world and is still under development. When Ericsson decided to go into the wireless telecommunication area, they built their platforms (NMT, GSM, etc.) based on the AXE system. Of course, modifications were made, but the main concept of AXE was still there [I1].

It has gone many years since the AXE system was born, and they use the same programming language. When a system is maintained, it gets larger and larger formatting. Maintenance will be harder and the source code in the system can get confusing. Confused means that the flow in code is hard to follow and it is difficult to understand the functionality. Different methods exist to solve the problem, but since the programming language that Ericsson uses when developing services in their systems has its origin in the 1970's, a programming language change may increase quality. By increasing the quality we mean lesser defects in both released and under development systems. Today "clean-ups" (rewriting code to remove the confusing part) has to be made after that about two or three projects have modified existing code. This is expensive, both in time and money.

Ericsson has found a programming language that they want to investigate for further use in new projects, SDL (Specification and Description Language). SDL has many benefits compared to their old language that they use today, PLEX-C (Programming Language for EXchanges, C-version). PLEX-C is a real-time programming language that Ericsson self has developed. It looks a little like Pascal, but the differences are many. SDL is graphical, with other words one "draws" the programs with help from a tool. The graphical interface makes the source code more understandable and easier to overview. Other benefits are



shorter development time and thereby lower development cost. The quality will be increased due to a more structured and formal development model, i.e. no manual coding and testing at higher level of abstraction earlier in the development process. Ericsson also has some parts that are designed in the programming language C, and in SDL both PLEX-C and C code can be represented at the same time (SDL is platform independent). More benefits are described in section 5.3.

Changing programming language is not easily done. PLEX-C and SDL does not have the same level of abstraction, i.e. SDL is a high-level programming language and PLEX-C is a low-level language. This is not the only reason for the complexity, differences in variable formats, signalling and not to forget the real time requirements. Time critical functions can in PLEX-C be written in the assembler language that PLEX-C code is compiled to, ASA, for optimizing the specific function. This must also be converted correctly. (Assembler is a programming language that is hardware dependent, i.e. each assembler statement corresponds to a single machine instruction.)

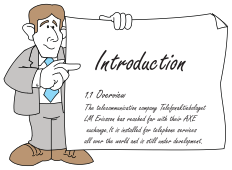
Converting from one abstraction level up to a higher is called *reverse engineering*. The reverse engineering area is large and increases fast. Much research effort is put here because, among others, many systems were developed during the 1970's and the designers did not think about the millennium change 25 years ahead. Reverse engineering can be used to solve problems related to the year 2000 problem (see section 3.6.1). Other things that reverse engineering is used for are clean-ups and maintaining systems. Ericsson uses clean-ups on their systems today, but since they cost more and more (the systems grow) and the fact that Ericsson uses an old programming language, make the profit for changing language higher.

1.2 Our work

If Ericsson changes their programming language, they cannot manually convert all old source code into SDL, it would take too long time and cost to much. Instead this has to be done automatically with help from a *reverse tool*. To see if the reverse tool is feasible, the SDL code is compiled back to PLEX-C code and there tested with the same test rules that is used for normal development in PLEX-C (see figure 1). Defects found here can have their origin either from the reverse tool or from the compiler (SDL to PLEX-C). Since the compiler is tested before, and there are test possibilities at SDL level, separation of the two kinds of defects will not be difficult. Defects from the compiler are not of interest for our work.

Even though that the reverse tool is working correctly, it is no guarantee that all parts in the system can be converted. If this is the case, these parts have to be converted manually. The parts that are impossible to convert automatically and why, are of interest for our work.

We have in our work focused on the consequences of the differences between the two languages (PLEX-C and SDL), i.e. what is difficult to convert and why. Since the differences are not represented in one single block, two blocks with different purpose have been converted.



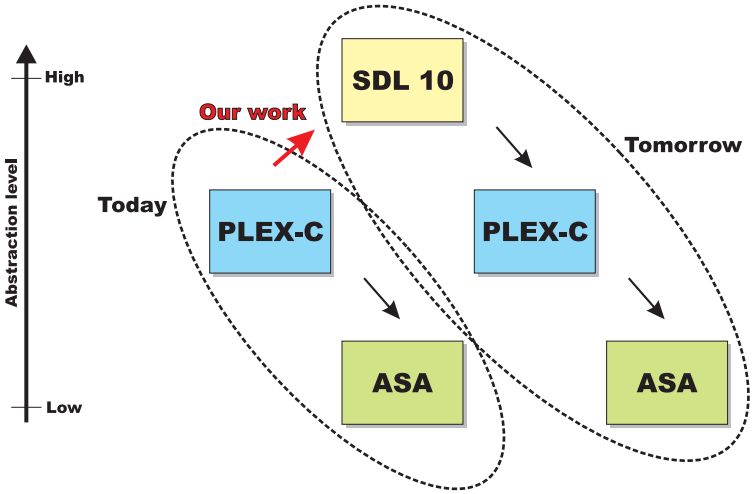


Figure 1. Our work in the software development

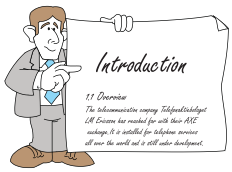
1.3 Organization

In **chapter 2** the questions that our work discusses are stated. **Chapter 3** briefly describes reverse engineering and other related work. Projects similar to our work are described too. In **chapter 4** and **5** the two programming languages PLEX-C and SDL10 are described. It is between these two languages transformations are done. **Chapter 6** describes and discusses the similarities and differences between PLEX-C and SDL. It is the differences that are of interest for our work. **Chapter 7** describes the tool that we have used in our work, and the major problems that we found when reverse engineering. In **chapter 8** our conclusions are stated, and future work is discussed. **Chapter 9** is acknowledgments and **chapter 10** is references.

Appendix A describes an overview of Ericsson's implementation of the GSM system, CME20. Our work is tested deep down in this system and exactly where is also described here. In **appendix B** abbreviations found in this thesis are described. **Appendix C** describes a process for how to make a conversion project.

1.4 Reading guidelines

Depending on your knowledge of the different subjects in our work, not all chapters are necessary to read, or not of interest for you. The reader must be familiar with the substance of chapter 6 to understand some parts in chapter 7 though, and the process script in appendix C is written on the basis that the reader has knowledge about Ericsson's development process and terms belonging to it.



1.4.1 Designer

The designer is the person who's interest is executing the process described in appendix C. Interesting parts for him/her may be chapter 3 for some background knowledge of reverse engineering and what different types of conversions that have been done before, chapter 4 and 5 if he/she has limited knowledge of the programming languages (PLEX-C and SDL), and chapter 6 and 7 to understand the problems that may occur while executing process.

1.4.2 Test engineer

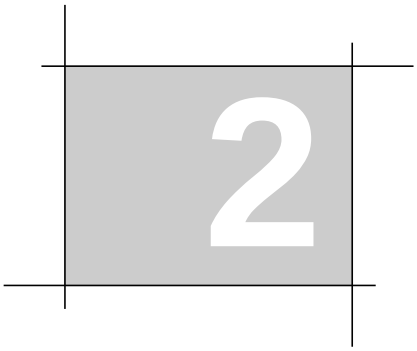
The tester is a person that test the interrelationships between blocks. To write good test cases he/she could be interested in reading the following chapters. Chapter 3 for background knowledge about reverse engineering and what that involves, chapter 5 if the tester does not have knowledge of the SDL environment (he/she is probably well familiar with PLEX-C and does not need to read chapter 4), chapter 6 and 7 may be interesting because of their description of the differences between the two languages and problems that may occur when executing a conversion.

1.4.3 Reverse tool purchaser

If the reader only is interested in the reverse tool evaluation, chapter 2, 6 and 7 could be useful reading. Lack of knowledge within the programming languages, chapter 4 and 5 may also be interesting reading. By reading these chapters, he/she will understand why we executed our work and gain knowledge about executing a programming language change process.

1.4.4 Quality manager

A quality manager may be interested in chapter 2 to understand why our work was executed, chapter 3 to gain knowledge about reverse engineering, chapter 4 and 5 to understand the two programming languages, chapter 6 and 7 to gain knowledge about the problems that a conversion brings, and finally chapter 8 to achieve our conclusion about reverse engineering source code.



Problem statement

2.1 Background

Today PLEX-C is used as programming language when developing and maintaining parts in Ericsson's CME 20 system (Ericsson's implementation of the GSM standard, see Appendix A). The language is old and today there are several other languages that fulfil Ericsson's requirements. One of them is SDL. SDL has many benefits towards PLEX-C, and they are described in section 5.3.

Changing programming language is not an easy task. There are many differences between the two languages that must be considered. The conversion is not made for the whole system at one time, instead parts called blocks are converted one at a time. Blocks can be divided into different categories depending on their task. To convert a block, a conversion tool, that handles most of the problems, will be used. But some questions still remain.

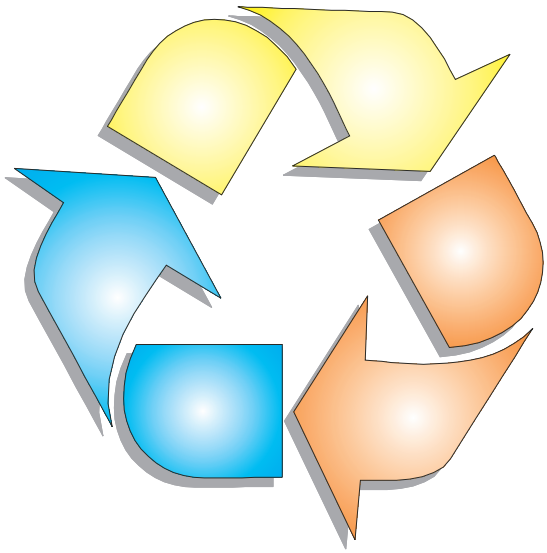
2.2 Problems

These questions are discussed and answered in this thesis:

1. **Is it possible to apply reverse engineering on PLEX-C blocks?**
   The tool that we will use for the conversion, does it work properly, or has it defects? If there are defects, how do they affect the conversion?
2. **If it is possible, on what type of blocks can we apply reverse engineering?**
   Since the blocks can be divided into different categories depending on their task (described in section 7.1), maybe all blocks not are convertible. If there is a difference in convertibility, what are the reasons for this?

3. **How long time will it take to convert a special block?**
   Is it difficult to convert a block, and how automized is it? Are there differences between different kinds of blocks, or is the converting time just depending on the size of the block?
4. **How is the reverse engineering applied to blocks?**
   This question will be answered with a process script that explains step by step how to do when converting blocks from PLEX-C to SDL.

Notice that the conversion is only done block by block and how they are connected to each other is not a part of our work.



3

Related Work

3.1 Reverse engineering

When developing everything from small programs to large systems, the development process can be divided into different phases. A process may consist of the phases requirements, design, implementation and test (see figure 2). This is a natural order for development of anything, because first one thinks of the products overhead functions, and later on more and more on detailed specifications of how the functionality will be implemented. This is called *forward engineering*, i.e. go from a high level of abstraction to a lower level.

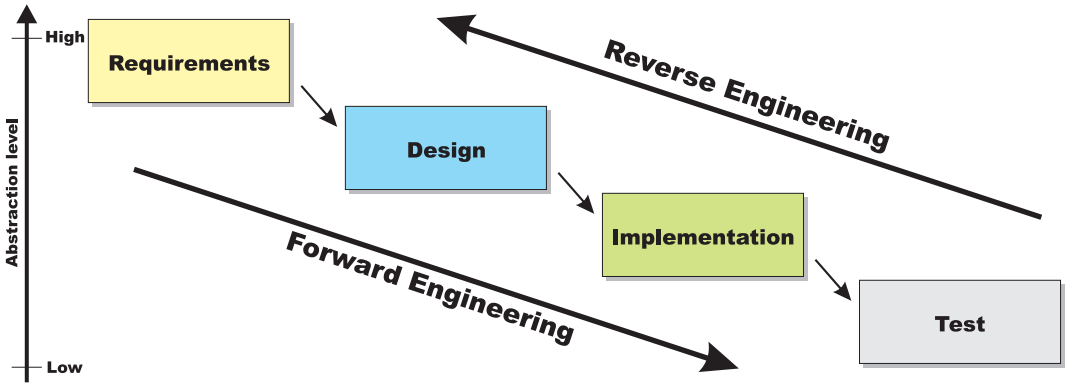
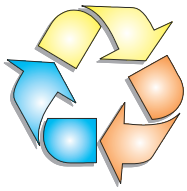


Figure 2. Forward and reverse engineering

The opposite to forward engineering is *reverse engineering*. The purpose of reverse engineering can simply be described as taking a product apart to learn how it works, or in other words, study a system and make a specification of it at a higher level of abstraction. One



example is making a design-document from C code. Reverse engineering is a term that refers to an analysis process which is done with help from methods and tools that investigate the system, its components, and their interrelationships. *Program understanding* or *program comprehension* are two other terms for reverse engineering that say more about what it is.

The origin of the term reverse engineering comes from the hardware technology, where it was used, among others, to duplicate other companies hardware products. Now this has shifted to software since it has become a larger part of whole systems with both hardware and software. In [P12] M. G. Rekoff defined reverse engineering (denoted to hardware technology) as

> "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system."

Five years later Chikofsky and Cross wrote [P4]:

> "Reverse engineering is the process of analyzing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form at higher levels of abstraction."

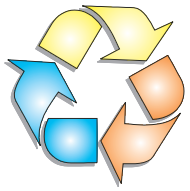
The two definitions above say exactly the same thing, but about different domains. This shows how close the domains are and similar methods can thereby be used when reverse engineering in the software world as in the hardware. In the hardware world reverse engineering is traditionally used to duplicate systems, while in the software world it is used to raise the abstraction level. Our work is within the software domain.

There are many reasons to use reverse engineering, and some may be [P11]:

- Improve the quality of one's own products.
- Analyze competitors' products to achieve knowledge of some of their secrets.
- Discover hidden defects in a newly developed product.
- Gain basic understanding of a system and its structure.

3.2 Methods

Reverse engineering can be seen as a set of methods and tools. The methods describe how reverse engineering should be done and the tools do it. Figure 3 shows an overview of the reverse engineering process. Information is retrieved from the source code by parsing and scanning it. The information is stored in a database in an organised way. Compiling the source code may also provide valuable information. The information is hidden in the object code and the cross reference tables (both from compiler), and is also stored in the database. Together, all information in the database describes the source code, and a new document can be produced. The document can present the information, or describe the source code, as the user likes, for example the description can be textual, flow charts, dia-



grams, etc. The important thing is that it describes the source code at a higher level of abstraction, which makes it to a design document.

In figure 3, the dashed lines corresponds to “information from” and not raw data, which the normal lines represent.

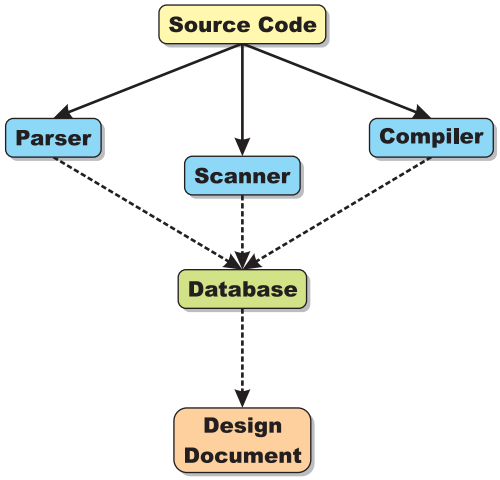


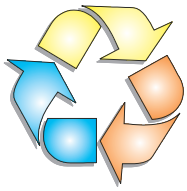
Figure 3. Reverse Engineering - The process and abstraction differences

A method corresponds to how the source code is investigated (parser, scanner, compiler), how the information achieved by the investigation shall be presented, the database structure (object oriented or not), and how the database should be investigated to produce desired design documents [P11].

The simplest method is light examination of hardware. The engineer investigating the product may not even know that he is reverse engineering. During the investigation he builds models of the product and how it works. The models may be notes, diagrams, or just mental images and plans. This method, as many of hardware corresponding methods, have no tool support. Tools are simpler to develop and implement in the software domain, and therefore the methods there are more sophisticated.

3.3 Related activities

Put reverse and forward engineering together, and add some purpose for applying them, then four activities can be stated. Except for purposes, the differences are denoted to the size of the reverse respectively forward engineering part. For example, the reverse part can be so small that the designer can hold the information in his head, or as large as several design documents. Since the differences between the activities are more on the purpose plan rather than practical, it is hard to make sharp lines between them. Chikofsky and Cross have described the activities in [P11].



3.3.1 Redocumentation

Redocumentation is the simplest and oldest form of reverse engineering, and can also be described as a weak form of restructuring (see below). The difference, towards reverse engineering, is that redocumentation does not change the abstraction level, instead it produces new representations of the system considering other point of views.

The intention with redocumentation is to improve the comprehension of a system and create additional views that were not created in the original forward engineering process. Redocumentation tools present facts about a system in another form, but without migrating between development phases. Examples of redocumentation tools are *pretty printers* (displays a code listing in an improved form), *diagram generators* (creates charts from code by reflecting control flow, code structure or data structure), and *cross reference generators* (produces index over the variable use in the program).

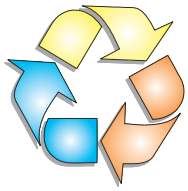
3.3.2 Design recovery

When reverse engineering, only the system itself is input to the process, but by adding existing design documentation, personal experience and knowledge about the problem and application domains to the input, fully describing documents at higher level of abstraction can be produced. This is called design recovery. The intense is to reproduce information required for a person to fully understand what the system does, how it does it, why it does it, etc.

3.3.3 Restructuring

Restructuring can be seen as a more advanced form of redocumentation or a special form of reengineering. The latter when reengineering without adding new functionality. The difference compared to redocumentation is how the redocumented, or restructured system is presented, and if the new version is presented in a different way according to its origin, the activity is restructuring. The new version is usually at the same level of abstraction as the origin, and the semantic behaviour and the functionality is the same, i.e. no new functionality or changes are provided when restructuring. If SDL and PLEX-C had been on the same level of abstraction, our work would be about restructuring instead of reverse engineering, but since SDL has a higher level of abstraction than PLEX-C, reverse engineering is the right terminology for our work.

Restructuring is done to improve a systems structure and make it more understandable. Often the term is used as a synonym for reproducing a program from an unstructured form to a more structured form (code-to-code). This may be transformation from “spaghetti code” (lots of goto statements) to more structured code with less goto statements. But the term has a broader meaning, for example data normalization, which is a data-to-data restructuring transformation and is done to improve the logical data model in the design process.



Restructuring can also be performed without knowledge of a systems structural form and without understanding its meaning. An example of this is a conversion of a series of if statements into a case statement, or vice versa.

3.3.4 Re-engineering

Reengineering is restructuring with functionality changes implemented. Actually it is a combination of forward and reversed engineering. First, reverse engineering is applied to gain more knowledge of the system and make design documents. New functionality and changes to the existing can then be applied. Finally, forward engineering is used and the system is re-implemented with the new/changed functionality.

3.4 Tools

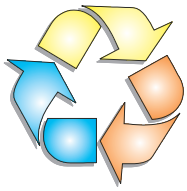
Reverse engineering methods are constantly developed to be applied on systems for different reasons. To make it as easy as possible for the engineer, corresponding tools are developed as help and guidance, or to do parts of the reverse process. As with methods, tools can handle one or several parts shown in figure 3, often several. Actually it is only one of the parts that is developed as a single tool, the compiler. A reason can be that its origin is not within the reverse engineering domain, but in the forward engineering.

As with the methods, many different tools for different purposes have been developed. The compiler tool is the only tool that corresponds to a single symbol in figure 3, and the other may correspond to several symbols including dashed lines.

More information about the tools described here can be found in [P9], and more information about CASE tools can be found in [P11].

3.4.1 Compilers

This tool is the most used reverse engineering tool today. There are several reasons for this. Compilers must understand the source program well enough so the compilation not change the programs functionality, if the compiler has an optimize function, it must understand the source program even better. Optimizing changes are larger and more complex than normal compiling. Some compilers can also understand what type of faults that the designer has introduced in the code and may also suggest solutions, some generates cross reference tables, warnings of portability problems (problems according to different target machines) and not initiated variables that may cause errors. Most of the compilers also have a debugging function. To support all this, the compiler must know the source program very well.



3.4.2 Restructurers and beautifiers

The purpose of using restructurers and beautifiers is to improve the comprehensibility of a system. These techniques are used on older programs or programs written in an old version of a software programming language that maybe not have constructs that exists today, for example the while loop construct does not exist in early versions of Fortran [P11] and are implemented with goto statements. By restructuring the program so it is implemented in a later version of Fortran, the goto statements are replaced. Goto statements are today “forbidden” and recognized as “spaghetti programming”. By replacing them, the program gets more understandable.

Beautifiers are a little more complex than restructurers since they also know the coding standard, i.e. layout rules, that the user wants. This can for example be indentation, bracketing conventions for compound statements, spaces in expressions etc. Beautifiers are used when standardizing the layout within large systems that have had small amounts of maintenance, and/or numerous of designers that works with different coding standards.

3.4.3 Translators

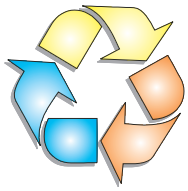
Translators are tools that convert source code from one software programming language to another, for example the PLEX2SDL reverse tool. If the languages are at the same level of abstraction (optimizing) the most common is that the new code is less readable, but if the new language is at a higher level of abstraction and the conversion is successful, the tool can produce a more understandable and modular program.

3.4.4 Parallizers

Parallizers are applied on programs that will be run on parallel computer systems. They make the code more effective by, among others, replace loops with single statements that simultaneously work on several elements in an array. The result is smaller and more readable source code which is specialized for a certain hardware.

3.4.5 CASE tools

As with compilers, CASE tools were first developed to support developers within the forward engineering domain. CASE is the abbreviation for *Computer-Aided Software Engineering* and represents a set of products, services and technologies for software development. CASE is for software engineers what CAD (Computer-Aided Design) is for constructors (e.g. ship-, space-, aircraft-constructers). The reason for developing such a tool is based on that not much effort were put in documentation and designing in the early years of 1970’s. Software systems were growing fast and more control over their development were desired and concurrently the faults had to decrease since they were the most cost inefficient part in the development phase. This concluded in that more and more effort were put earlier in projects. Methods saying how to run software development projects were “invented” and tools helping designers follow the methods were also produced



(CASE tools). Since then CASE tools are getting more and more efficient and gets more and more functionality.

At the beginning of the 1990's, it was discovered that CASE could be used when maintaining systems, and thereby also when reverse engineering systems.

A CASE tool consists of a graphical editor, consistency checkers and may also have a code generator. The graphical editor is used to present the high level design graphically and not textual. The consistency checker is used to test the system and the code generator can produce the code, or at least a part of it.

CASE tools that have the ability to reverse engineer systems can in most cases produce the graphical pictures of the high level design automatically. These pictures can represent different diagrams depending of what knowledge one wants about the system. Since this is a topic in great research, more diagram types and further analysis will come in the future. Some types of diagrams can be:

- **Structure chart**
  This diagram is the most common today and shows the subprograms and how they are connected to each other.
- **Data flow diagram**
  This diagram shows the major software modules and data allocation, and how data and control information flows among them.
- **Entity relationship diagram**
  This diagram describes major external sources, data and modules that uses them.

3.5 Balance between reverse and forward engineering

It is expensive to reverse engineer a system, but it is also expensive not to do it. The maintenance cost is increasing as the system gets older, because the maintainers must understand the system to be able to maintain it. This means that ideally a defined balance between forward and reverse engineering has to be found. van den Brand et. al. has taken a closer look at program development regarding forward and reverse engineering [P3]. Figure 4 shows different phases of the life cycle for a software system, **r** and **f** are a measure for the reverse respectively forward engineering effort. **a** shows the classical life cycle and **b** the desired life cycle of software systems.

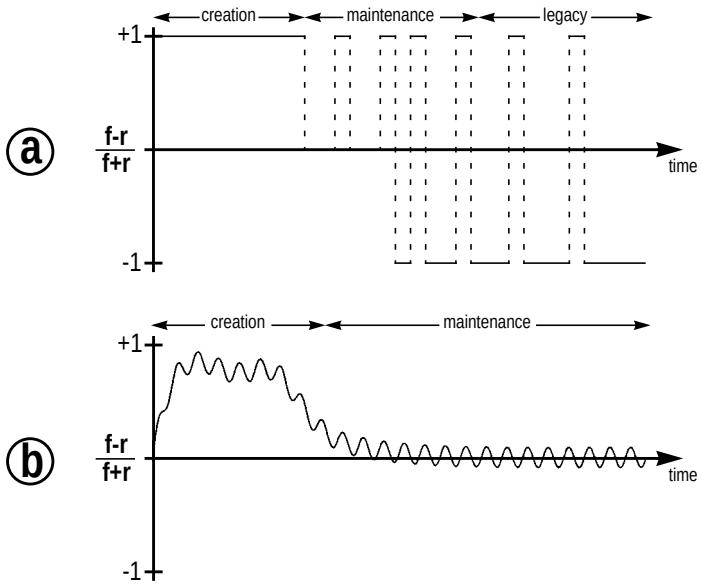


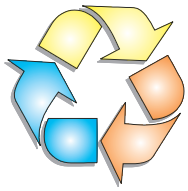
*Figure 4. Differences between **a.** classical and **b.** harmonic software engineering*

Creation
a. Only forward engineering is used in this phase, which is the classical way when developing software systems.
b. Reverse engineering is used directly at the start of the development, combined with forward engineering, to influence the design. This could for example be to study the impact of different implementation alternatives. One can say that this is a reverse engineering driven software development.

Maintenance
a. To keep the system running small amounts of maintenance is needed, and the amount will increase. The forward engineering is interrupted by larger and larger reverse engineering periods.
b. As in classical software engineering, maintenance is needed in the harmonic software engineering to keep the system running. The knowledge about the system is kept up to date with reverse engineering, and maintenance is done with the forward engineering.

Legacy
a. When the reverse engineering takes to much time, the maintenance is hard to keep up. The maintainers are working more on understanding the system instead of maintaining it.
b. The reason of harmonic software engineering is to skip this phase, and thereby have a product with lesser defects and longer lifetime.

By using the facts that van den Brand et. al discusses in [P3], better systems that are easier to maintain will be produced. Reverse engineering will also be a larger part in software engineering in the future.



3.6 Practical use of reverse engineering

Many reasons why reverse engineering should be applied on a system exists and in this section both reasons for reverse engineering a system and reverse engineering projects are described. A strong connection exists to related activities described in section 3.3.

3.6.1 Year 2000 problem

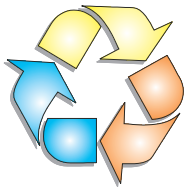
In many computer systems, only two numerals are used two define a certain year, e.g. 87 for 1987, and some of these systems uses this numerals to count a difference between two years. This is no problem as long as the numerals represent years within the same century. Let us say that you want to calculate someone's age. You know the year of birth, 1946, which is represented as 46 in the computer system. To calculate the person's age, 46 is subtracted from today's year. The age will be 99 - 46 = 53 years. Next year (2000) the equation will be 00 - 46 = -46, which is an incorrect answer. In some systems negative numbers cannot be stored and a failure will occur. If so the user of the system is alerted and can correct input data or calculate the data himself. But if a wrong value is stored and used in further calculations, final output data will be corrupt and that is not always discovered by the user. Other faults that may occur can be wrong decisions made by the computer system which will conclude in for example that railroadpoints are set wrong (how and if the year is involved in railroadpoints or not, is not a part of this thesis).

The year 2000 problem is complicated. It is hard to find and can be found in many systems. Many companies put major resources to solve this problem. By using reverse engineering to raise the abstraction level of a system, knowledge about what parts that may be affected by the millennium change can be achieved. These parts can then be re-designed so they can manage the problem. But the largest problem is the cost in money, personnel and especially time, it has to be solved before the turn of the year 1999 - 2000.

3.6.2 Data reverse engineering [P8]

The reason for applying reverse engineering at systems to solve their year 2000 problem is done because one wants to get data about the data inside the system. This retrieved data is called metadata (see section 3.6.3). The metadata is not only used to solve the year 2000 problem.

Today, with rougher competitions between companies, each company must achieve as much valuable information as they can from data that they already have. This data can be statistics over customers etc. The problem is that many companies have lots of data, but they do not know that these data holds valuable information for them. Example of such data can be what, when and how much customers buy, if the customer pays with a credit card, etc. The data can be stored in different ways, for example locked in systems whose designers retired long time ago, in applications that were produced as a temporary fix, but have been in use ever since or even as data that the organization does not know that it has.



Organizations need to have the right data and it is important that they also know what it has, were to find it and most of all know what it means. To retrieve the valuable information from the data, one can apply reverse engineering and collect metadata. The metadata helps the organization to understand it's data.

3.6.3 Data conversion [P7]

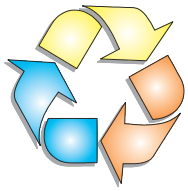
The departments of Personnel and Training (DP&T) and Accounts (DOA) at the Commonwealth in Virginia wanted to replace their existing payroll and personnel information systems because they had become inefficient, too expensive to operate and maintain, and the management were concerned about keeping the staff up to date with the technical knowledge of the databases. The system consisted of two large databases and software to manage them. The databases were not integrated, which the departments wanted so they could merge payroll and personnel records. A new system consisting of three modules from *PeopleSoft*, was decided to replace the old one. PeopleSoft's modules are build specially for server - client applications and should be tailored to fit each organizations need. This is what the departments wanted. The problem was to move the data from the old databases to the new ones. It was a large amount of data to move. A tool that could convert the data had to be developed. To know how to convert the data, more information about the databases, both old and new ones, were needed. This information (data about data) is called *metadata*. To get the metadata reverse engineering were used. To manage all metadata that this project produced, a new database and handling procedures called *The Metadata Access Tool* (TheMAT) was developed. It uses Microsoft Access with both automated and manual procedures. The metadata were later on used to map corresponding information in the two systems (the old and new).

This restructuring project ended successful and the conclusions were that it was not so expensive as expected to use metadata for developing data conversion tools and the metadata itself can be maintained easily with tools. Some metadata can be maintained by using a CASE tool (see section 3.4), especially if such a tool were used when developing the system. Metadata that easy can be created and maintained can be a valuable asset.

3.6.4 PL/IX - C++ [P6]

Reverse engineering, or rather restructuring, is a useful activity when changing software language. In a project funded by IBM and project members from universities of Waterloo, Victoria and Toronto, PL/IX (pronounced PL nine) source code should be converted to C++ code using reverse engineering. PL/IX is a programming language that IBM uses and to simplify maintenance and make further development of the system easier, IBM wants to change software language. From the Universities point of view the project was an experiment to see how hard it is to make a software language converting tool.

To compare and to be able to map constructs (if, for, etc. statements) and structures (variable types) between the two languages an abstract syntax tree (AST) were produced with help from a custom built PL/IX parser and linker (compare with compiling). Defined con-



structs and structures were sorted into different domains. If one construct or structure in PL/IX did not have a corresponding item in C++, a new one had to manually be added to the domain library.

In the first half of the project the converting was manually performed and in the second half a semi automatic tool was used. When the project was finished, the tool still was semi automatic and some parts had to be converted manually. The conclusions from this project was that it is possible to develop tools that decrease the manual effort in a software language change.

3.6.5 SDL Reverse [I4]

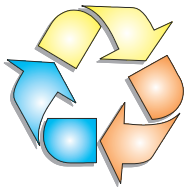
When Ericsson choosed to investigate the possibility to change software language in their GSM system from PLEX-C to SDL10 they soon realized that it was impossible to change all source code manually. A conversion tool was required. A project started to develop such a tool, *SDL Reverse*. Unfortunately the inexperience, the bad estimations and mostly the fact that the task actually is hard to manage made the project fail. Instead another company took over the development of the "*Reverse tool*". They are now almost finished with it and it is this tool that we have used in our work.

This project's purpose was to develop a tool which reverse engineers a system. The project itself is about forward engineering, but inside the produced tool the activity can be classified as either reverse engineering or re-engineering, depending on the point of view. The tool reverse engineers the system to achieve information about it and then implements the information in another software language. The information is at a higher level of abstraction than the new implementation, and forward engineering is used. Merging the processes concludes in re-engineering. But from another point of view where the tool is seen as a black box performing the transformation, the activity is reverse engineering.

3.6.6 KomPlex [I5]

Our work is not the only project that have tested the reverse tool, the KomPlex project, finished in the summer of 1999 at Ericsson's department in Aachen, Germany, has also tested it. The project's purpose was to evaluate and test the reverse tool and see what kind of blocks that could be converted, but also to evaluate if Ericsson should continue with reverse projects and thereby make a software programming language change. Problems with the tool was reported during the project to the producer, and some of them were fixed during the project.

The project was intended to convert eight blocks but they tried out 21 blocks where 16 were successfully converted and 5 failed for different reasons. A conversion is successful when it passes a test that tests at least 80% of the code. The failure reasons can be unimplemented support in the reverse tool for some kind of constructs in PLEX-C. A connection can be seen if the blocks are divided into groups. The division is made on basis of the



blocks' tasks since the structure of a block depends on its task (see section 7.1). The categories are:

- Traffic blocks
- Message handler blocks
- Database blocks
- Command blocks

To analyse and see if successfully converted blocks can be accepted in the system, more aspects than testing the code has to be done. Measurements of the new code size (number of lines), the data storage size (e.i. the variables total space) and effectiveness (execution time) were done. On the basis of these measurements the following conclusions have been reached.

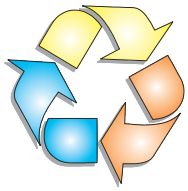
Traffic blocks seemed to manage best. The regenerated code size and data storage, and execution time were all within the 10% characteristic limit. The regenerated code from *Message handler blocks* increased with 17% which not is acceptable. The execution time is just above the limit (11%). The reverse tool will be updated to handle this problem. *Database blocks* were also slightly above the limit, but optimizations in the SDL2PLEX generator will solve this problem in the future. The block category that was hardest to manage were *Command blocks*. Only one block were possible to convert successfully but the same tendencies could be seen in the unsuccessfully converted blocks. The data storage increased with over 100% and regenerated code size became 50% larger. Although these blocks are small compared to other blocks and they are not so time critical, also updated reverse tool to decrease these figures will make it possible to convert this blocks in the future.

The conclusion from this is that after updates to both the reverse tool and the SDL2PLEX generator mostly all blocks will be able to convert.

Also a small formula for estimating how long time it will take to convert a block has been produced. It considerates only the number of PLEX-C statements to convert though. This formula is tested within our work.

The members of the project had different experience, some PLEX, some SDL and some both. This composition of people with different knowledge appeared as a good prerequisite when running such a project.

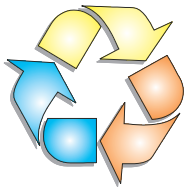
The KomPlex project has answered most of the problems that our work is bringing up, but the structures inside different blocks at different departments within Ericsson are not the same and therefore cannot all results from one project just be copied to another. Of course some considerations of the results will be made, but own experience at the departments are needed. The KomPlex project was also larger with both more personnel, more time and more blocks to convert than our work.

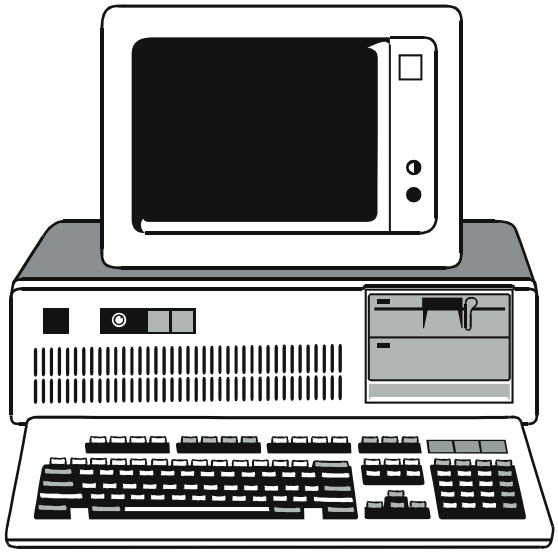


3.6.7 SPOT [I6]

To get some experience of converting PLEX-C blocks to SDL10 at Ericsson's department in Hässleholm, the SPOT project started in December 1997. The task was to convert a block into SDL10. The reason was to evaluate if a software language change could increase quality and decrease lead time for future projects. Also the SDL2PLEX code generator was evaluated according to capacity, memory size (data storage) and readability (the regenerated PLEX-C code must be readable for a human). Since the reverse tool did not exist at that time, the conversion was made manually. The conclusions were that all in the project thought that a software language change will shorten lead time and increase quality. The overall impression of SDL10 was good and that it was easy to use. The SDL2PLEX code generator on the other hand did not satisfy the expectations. Lots of improvements had to be made and a list of needed improvements were produced.

The differences between this project and KomPlex are small, both converted PLEX blocks to SDL10, but this project did it manually. Since a higher level of abstraction than the destination implementation were reached during the project, it should be classified as reengineering.





4

PLEX-C

4.1 History

The high-level programming language PLEX (Programming Language for EXchanges) was developed for Ericsson, by Ericsson in the 1970's, and later extended in 1983. The reason why Ericsson developed a new programming language of their own is simple - no other languages that fulfilled Ericsson's requirements existed. It had to be a high-level, real-time language with very strict requirements regarding real-time performance. The structure should be modular and the modules should communicate with different signals. All this is implemented in PLEX. PLEX is only used for telephony purposes, but it exists in thousands of exchanges all over the world, and it is also used by thousands of designers. PLEX is a company specific programming language, i.e. only one company uses the language in development, and that makes is hard to find documents describing the language. More information can be found in [I3].

4.2 Versions

PLEX exists in different versions, PLEX-M and PLEX-C. PLEX-C is used when programming the processor in AXE 10, and since CME 20 is built on the AXE 10 switch technology, PLEX-C is used when developing services here. PLEX-M is used when programming a special part in the AXE 10 system called EMRP, which controls the subscriber stage. PLEX-M is an 8-bit version of PLEX-C, which is a 16 or 32 bit programming language, depending on what processor the target system (exchange) has.

4.3 System

The code itself has similarities with Pascal, but the differences are many. The major differences are:

- only one variable type (a group of bits) which can have different properties
- different jump statements
- negative numbers cannot exists
- pointers are in reality circular array indexes and not memory addresses
- it is a real-time language, means that the order of execution is not predictable before execution
- communication between blocks are handled by signals

4.3.1 Standard

Many designers are involved when a product is developed, and with new releases, changes in the source code will be made. For that reason design rules exists. The design rules tells the designer how to implement the program so that other designers easily can make changes. When programming PLEX code, one must comply with a “PLEX standard”. For a system written in PLEX, five different kinds of documents have to exist and all of them have a common part, an ID sector at the end of the document. This ID sector contains information about the document, e.g. document number, author, responsibilities, version etc. The documents are:

- **SPI -** Source Program Information
- **SPL -** Source Parameter List
- **PL -** Parameter List
- **SS -** Signal Survey
- **SD -** Signal Description

SPI - Source Program Information
This document contains the source code. It is here the designer writes or changes the executable statements. It can be PLEX-C code but also ASA assembler code. ASA is used when time critical functions have to be implemented.

The SPI document consists of different sectors. The first is a *declare sector* in which all declarations are stated. The next sector is the *program sector*, which can in itself be two sectors, one for PLEX-C code and one for ASA assembler code. The third is the *data sector*, in which initial values are assigned to some variables declared in the declare sector. Also statements to specify the order of the variables in the data store are here. The last sector is the *ID sector* which is described above.

SPL - Source Parameter List
This document contains the default parameter values for all data in the PL document (see below).

PL - Parameter List
The products that Ericsson develops will be released in many different countries and certain data has to be adapted to the local market. This can for example be tone-types, printouts in the local language, charging parameters, etc. These market dependent data should be included in this document to avoid frequent modifications of the source code in SPI.

SS - Signal Survey
The signals that a block sends and receives are listed in a Signal Survey document. Each block has its own SS document.

SD - Signal Description
There is one SD document for each signal. The SD document contains a description of the signal, information about the signal's purpose, type and data. All SD's are stored in a signal-handling library.

All these documents together form a PLEX-C program. A PLEX-C program example (not all documents, only the SPI) is shown here. The program calculates the difference between a received value and a max value (set in the data sector).

```
DOCUMENT PROGEXAMPLE;
 DECLARE;
  VARIABLE CNUMBER (16) 4 DS;
  VARIABLE CNUM 4 DS;
  VARIABLE CMAX 4 DS;
 END DECLARE;

 PROGRAM;  PLEX;
  ENTER MYSIGNAL WITH CNUM;     ! RECIEVE SIGNAL !
  CNUMBER = 8 - 31;             ! CNUMBER = 65513 (NO NEGATIV NUMBERS) !
  DO SUM;                       ! CALL SUBROUTINE SUM !
  SEND YOURSIGNAL WITH CNUMBER; ! SEND SIGNAL !
  EXIT;                         ! SET PROGEXAMPLE IDLE !
 END PROGRAM;

 PROGRAM SUM; ASA210C;          ! ASA SECTOR, SUBPROGRAM !
  RS WR1-CNUM;                  ! STORE CNUM VALUE IN REGISTER WR1 !
  RS AR0-CMAX;                  ! STORE CMAX VALUE IN REGISTER AR0 !
  AR WR1-AR0;                   ! CALCULATES AR0 - WR1 AND STORE ANSWER IN WR1 !
  WS CNUMBER-WR1                ! STORE WR1 VALUE IN CNUMBER VARIABLE !
 END PROGRAM;

 DATA;
  SIZE OF CMAX = 10;
 END DATA;
END DOCUMENT;

ID PROGEXAMPLE TYPE DOCUMENT;
 CLA 19055;
 REV C;
 DAT 99-04-19;
```

```
 DES ERA/LVA/DX MBER;
 RES ERA/LVA/DC;
 APP ERA/LVA/DC;
END ID;
```

5

SDL

5.1 Description

SDL is the abbreviation for *Specification and Description Language*. It is a high-level, object-oriented and graphical language, which is intended for developing complex, real-time and communication systems. Examples are cellular and DECT phones, exchanges, radio systems, and train-control systems [W1]. A program written in SDL can be presented in two ways, either graphical or textual. The most common way is the graphical because of its benefits (see section 5.3).

5.2 History

In 1972 a study group within the telecommunication union CCITT (now called ITU-T) began to research on a specification language that the telecom industry could use. In 1976 SDL got standardized by ITU-T, as standard Z.100. Every fourth year a new version is released. The modifications in summary are [P2]:

- SDL-76 First standardized version. It only had recommendations on how process graph symbols should be drawn.
- SDL-80 The block conception is introduced and the PR-form (textual representation) becomes a part of the language.
- SDL-84 Additional concepts are introduced, among others the concept of abstract data types.
- SDL-88 Only minor changes.
- SDL-92 SDL becomes object-oriented.

- SDL-96 Minor changes, e.g. external procedures

5.3 Benefits

The benefits that SDL has compared to other programming languages such as C/C++ and PLEX are many. Some of them are [W1][P2]:

- **Graphical user interface.** The graphical interface makes the software easier to understand, even for a non-technician. One can easy get a clear picture of how the system is built up by different parts, and how they communicate.
- **Easy to use.** Designing a SDL program is done graphically with help from a tool. The SDL code can then be translated into executable code without any manual “line” coding, as in C++ programming. This makes the development time shorter and increases the quality.
- **Documentation.** Since SDL is graphical, the program itself becomes a document that is easy to read and shows how the system is implemented.
- **Test and maintenance.** The fact that SDL has a rich grammar which describes behaviour, makes it possible to build simulation tools for SDL systems and validate formal characteristic (e.g. to avoid deadlock). This means that defects can be found very early in the development process.
- **Design and implementation independent.** SDL is independent of the design paradigm, i.e. if it is function oriented (PLEX-C) or object oriented (C++). SDL is also independent of the implementation language, which means that SDL code can be compiled to any language one want, e.g. C, java.

5.4 Telelogic Tau

In our work we have used a toolkit for SDL development, and it is the Swedish company *Telelogic AB* that have produced (and now improves) it. The toolkit is called *Telelogic Tau*, hereafter referred to as Tau. Tau is an SDT (SDL Development Tool), and for that reason Tau sometimes are referred to as SDT, i.e. SDT and Tau means the same thing. With Tau it is possible to build SDL applications, test them, simulate live performance, debug SDL programs, make test files etc. Many companies, for example Telia, Siemens Defence, Atlas Copco, Alcatel, Ericsson, IBM, Intel and Nokia use Tau in their development [P2]. Tau is a set of several tools connected to each other as shown in figure 5.

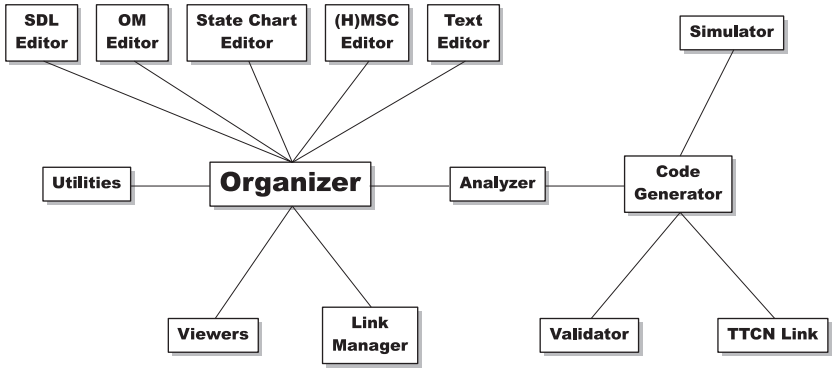


Figure 5. Telelogic Tau

Organizer
The Organizer is the central tool in Tau. When starting, the Organizer is appearing. It shows all components in the system and how they are connected. One can compare the Organizer with an advanced file manager.

Editors
Tau has several editors for different purposes. The most used editors are the *SDL Editor* and the *MSC Editor*. With these editors SDL and HSDL (High-level SDL) respectively MSC and HMSC (High-level MSC) can be edited. Other editors are *Text Editor*, *State Chart Editor* and *OM Editor*. State Charts show an overview of the states in an SDL program, and OM (Object Model) concerns objects in high-level design.

Analyzer / Code Generator
The *Analyzer* checks SDL systems for errors, such as unconnected symbols, undeclared variables, type conflicts, etc., and the *Code Generator* makes executable code. The code can be C code, PLEX-C code, or compiled code ready to run, for instance in the simulator or the validator.

Validator / Simulator
With these tools a system can be tested. The difference between them is that in the *Validator* MSC's are compared with the SDL code to see if their flows correspond to each other, and the *Simulator* simulates the program and a test engineer can send in any signal with parameters and check what he gets back. Of course both validation and simulation can be automated.

Viewers
The *Type Viewer* shows objects and their inheritance, the *Coverage Viewer* shows how much of the code that was covered by a certain test (with validator or simulator), and the *Cross Reference Viewer* is used to locate definitions and all references to them.

Utilities
An example of a utility is the *Preference Manager* where all settings for Tau can be adjusted.

Link Manager
This tool handles links between different objects in the system, where objects may be text fragments in text documents or graphical symbols in for example SDL and MSC diagrams.

TTCN link
TTCN (Tree and Tabular Combined Notation) is a standardized test language that makes it possible to test the system with same test files in different environments, and the *TTCN link* is what it says, a TTCN link.

5.5 SDLtool

Since PLEX-C and SDL does not support the same things (more in section 5.7), Tau has been adapted to fit Ericsson's need for development, and that tool is called *SDLtool*. The differences is that some local tools that help the designer with the differences between SDL and PLEX-C has been added, and some functionality, e.g. object orientation, is removed due to the fact that PLEX-C does not support them. A new code generator has also been added, a PLEX-C generator. It converts SDL code to PLEX-C code, and is further referred to as the PLEX-C generator, or SDL2PLEX. SDLtool is built on Tau, and new versions of Tau also make new versions of SDLtool.

5.6 The SDL language

An SDL system consists of several *extended finite state machines* (EFSM) that run in parallel [P5]. The EFSM is an extended concept of the *finite state machine* (FSM). The FSM consists of a set of states with the possibility to receive signals. The signal that is received sets the next state. No variables are allowed and that makes the FSM good for small problems only. Some functions, such as counting, will bring a need of many states. Therefore the FSM concept was extended to the EFSM. In this machine variables are allowed. The EFSM's, or processes, communicate with signals and runs independently.

5.6.1 Components

The following components are embedded in an SDL system [P2]:

- **Structure** System, block, process, and procedure hierarchy
- **Communication** Signals with optional parameters and routes (channels)
- **Behaviour** Processes
- **Data** Abstract data types (ADT)

- **Inheritance** Describing relations and specialization

Structure
An SDL system is divided into four hierarchical levels (see figure 6).

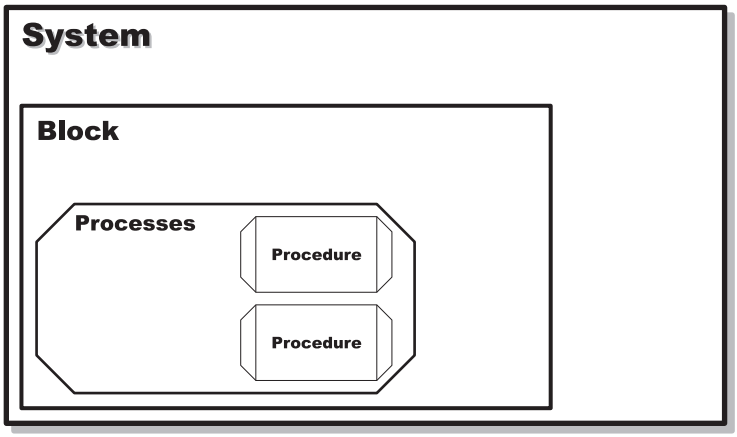


Figure 6. SDL system

- **Procedures** may be recursively implemented and they can both be local to their process or global, depending on their scope.
- **Processes** in SDL have its own separate memory space and is defined as a nested hierarchical state machine.
- **Blocks** are a set of processes and other blocks grouped together.
- **The system** is where all blocks are connected to each other and to the environment.

Communication
The communications inside an SDL system, between the processes, are made with signals. They are asynchronous, i.e. the order of their execution cannot in advance be stated. Remote procedure calls can be seen as synchronous signals, i.e. correspond to goto statements or subroutine calls. Both signal types can carry parameters to the receiver. It is not only SDL processes that can send and receive signals but also hardware (called environment in Tau) or non SDL applications. This is for instance necessary when using timers. An SDL process can set timers, and when the timer expires, a timer signal is sent to the process. The timer can also be mapped to an operating system timer or a hardware timer, which makes it possible to simulate time in SDL models, before the target system is available [W2].

The idea of SDL's clear signal interfaces between different parts in a system simplifies large team development and ensures consistency between the parts. But signals and processes cannot be prioritized, priority does not exist in SDL.

Behaviour
Processes can be created at system start and at run time. They can also be terminated at any time. More than one instance of a process can exist, and all instances have their own identification number PId, so signals can be sent to special instances of a process.

Data
Data can be described in two ways, abstract data types (ADT) and abstract syntax notation one (ASN.1). ASN.1 enables sharing of data between languages. ADT has no specified data structure, instead a set of values, operations and equations are specified. Standard variables are also available, such as integer (numbers without decimals), real (numbers with decimals), boolean (true or false), time, charstring (text), PId (process identification), etc.

Type declarations (the part where the programmer declares which variables he/she will use and what kind of types they are) can be placed any were. It can be either inside the system close to their context, at system level, or even outside the system in packages, which makes it possible to share declarations with other systems.

Inheritance
In object oriented languages one of the major benefits is that new objects can be created by adding new or changing properties to existing objects (specialization). This can of course be done in SDL, but since SDL10 is not object oriented (see section 5.7), this is not explained here.

5.6.2 The layout of SDL

To understand the graphical user interface of SDL and how easy it is to understand, figure 7 shows an example of an SDL process.

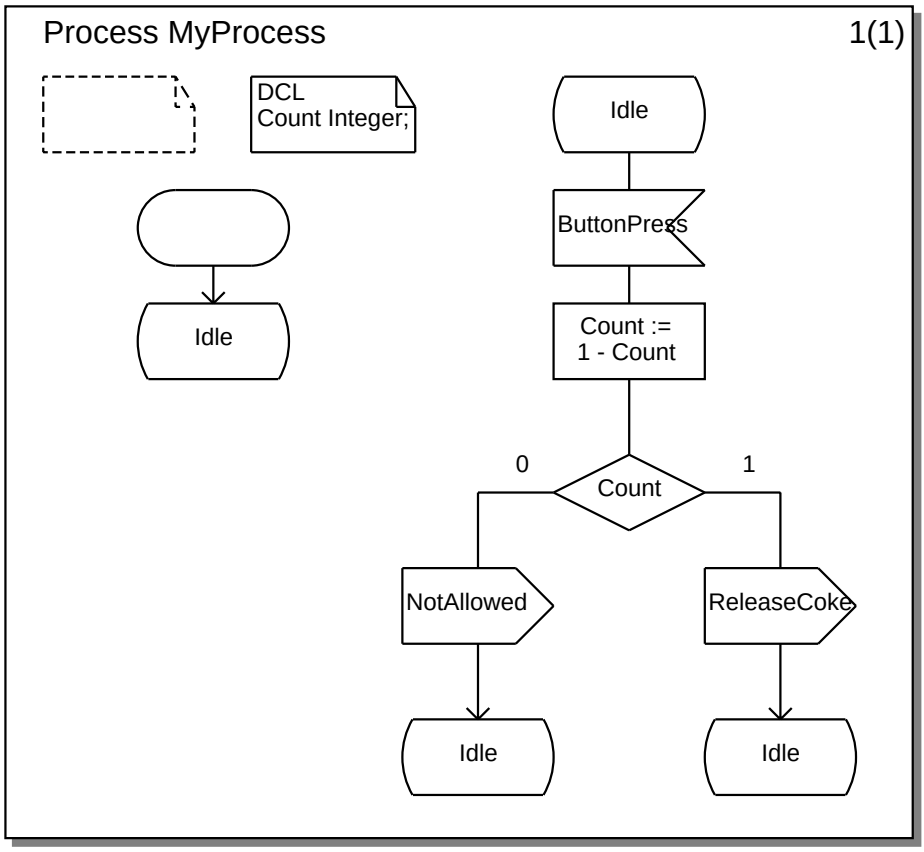


Figure 7. SDL Process

The process is “resting” in the `Idle` state, waiting for the signal `ButtonPress` to arrive to the process. When the signal arrives, a local variable, `Count`, is set to a value. In this case it is oscillating between 0 and 1 every time the signal arrives. Then the value that `Count` holds is checked, if it is 1 the signal `ReleaseCoke` is sent, otherwise, if `Count` is 0, the signal `NotAllowed` is sent. After any of the signals are sent, the process goes to its `Idle` state.

Where the signals are sent cannot be seen in this figure. It is the overlaying block views that show that, and block views will not be shown here.

5.7 SDL10

SDL10 is an adapted version of the standardized SDL language, made for fitting Ericsson’s need. All functionality that PLEX-C gives must also be supported by SDL, and with both limitations and extensions towards SDL, SDL10 fulfils their requirements. Tau supports

SDL, and SDLtool supports SDL10. Differences between PLEX-C and SDL are described in section 6.2.

The extensions are implemented in SDL as directives, i.e. direct commands to the SDL2PLEX compiler. The disadvantage of this is that it will be harder to test the systems behaviour at SDL level. Directives are written as comments in SDL, and begin with a keyword (called directive). The directives can have parameters as well. Examples of extensions are ASA subprograms, external code, ID sector, signal priority and temporary variables.

Limitations are functionality in SDL that are not supported in SDL10. An example of a limitation is the object-oriented concept and thereby also specialization.

5.8 MSC

In real time systems the different processes performs tasks and like all programs they need input data to make decisions (otherwise the developer should know the result of the program execution). A process retrieves data from other processes or components, such as the environment, and it is carried on signals. Signals, or events, must happen in a specified order, e.g. it should not start to ring in a phone before the caller has finished the dialling. An MSC (Message Sequence Chart) shows chronological sequences of messages, or signals, sent between components and their environment. MSC is like SDL standardized by ITU-T, Z.120 [P2].

An MSC is useful for describing the dynamic behaviour of a system. The graphical presentation shows complex behaviours clear and it is easy to understand. Even non-technician people can understand MSC’s.

In Tau, test files for SDL systems can be generated from MSC’s. An MSC does not describe the complete behaviour of a system though, rather one execution trace. On the other hand several MSC’s can describe the system more detailed [P2].

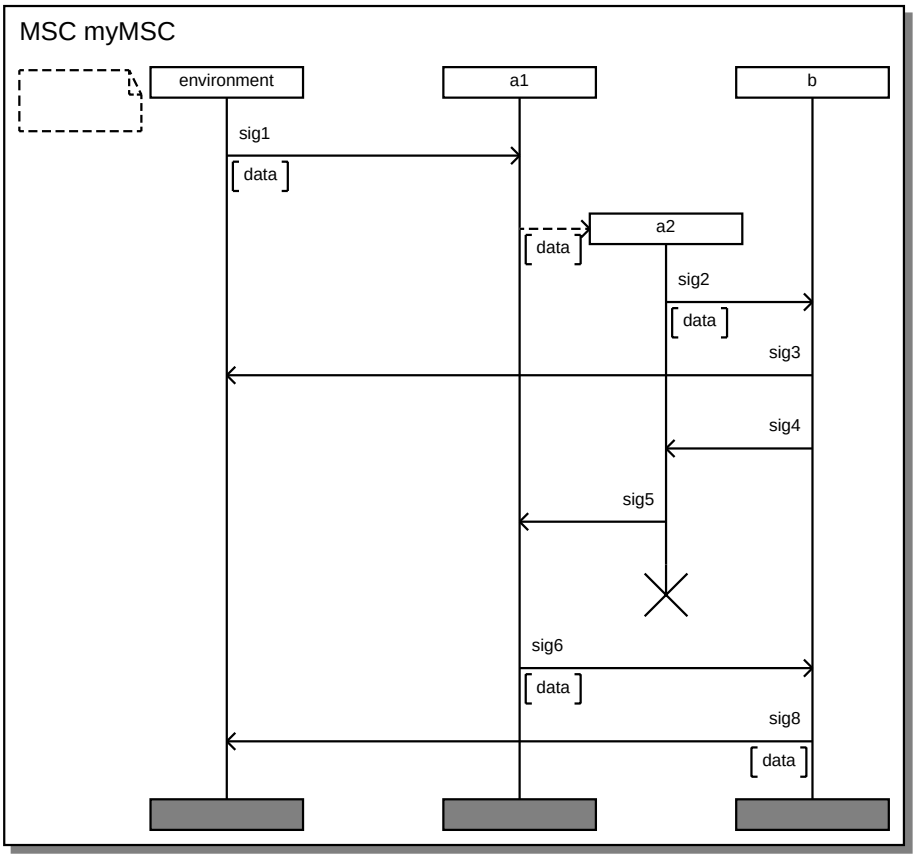


Figure 8. MSC

Figure 8 shows an MSC. The rectangles at the top with text inside are different processes or components in the system. When they are at the top it means that they are already started and are running when the events that this MSC shows begin to happen. The grey boxes at the bottom means that the components not are terminated here and will be running after the events that this MSC shows. The dotted line marks creation of instances during run time in SDL. Process **a2** is created by process **a1**, and then terminated, showed with a large cross. The arrows in an MSC represent events, which is signals in SDL. The text above an arrow is the name of the signal and the text below, inside the brackets are the data sent along with the signal [P2].

6

Comparisons - SDL vs PLEX-C

There are both differences and similarities between SDL and PLEX-C. The similarities are the reasons for choosing SDL instead of any other programming language. The differences are the major problems for our work. To have a successful programming language change, we must find solutions for all the problems that the differences bring.

6.1 Similarities

Realtime
Both PLEX-C and SDL are realtime languages, and use signals to communicate between different parts in the system.

Development
Development in SDL, or actually in the SDL environment, is similar as in the PLEX-C environment. The tools are different of course, but their purpose are the same. Figure 9 shows the corresponding documents between SDL and PLEX-C. Dashed lines represents automatic steps. Note the differences in the abstraction levels.

- First MSC's / Sequence diagrams are produced. They both show interaction between blocks, i.e. signals sent and received within certain functions, but sequence diagrams explains a little bit more than MSC's and have therefore a lower level of abstraction.
- The next step is generation of SDL code respectively flow charts. They both describes what should be done between the signalcommunication described in the MSC's / sequence diagrams. Since the "what to do" part is implemented with statements in SDL and with explaining sentences in flow charts, there is a difference in abstraction level.

- In the last step PLEX-C code will be generated. In the SDL environment this is automatic, but in the PLEX-C environment this is a manual step and done with help from flow charts.

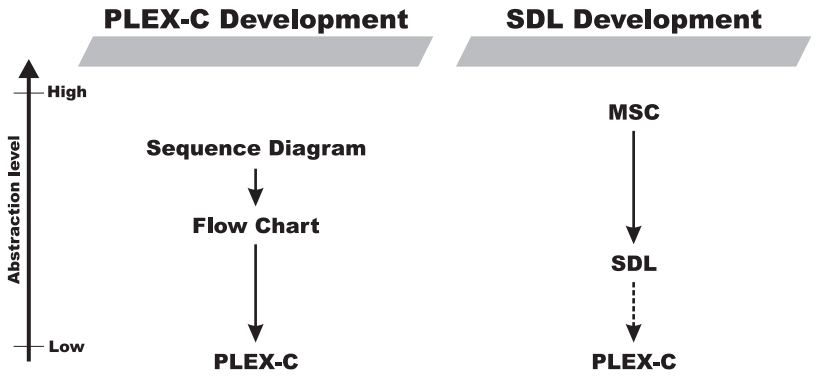


Figure 9. SDL vs. PLEX-C environment development

6.2 Differences

Variables
In PLEX-C the variables are a set of bits stored in the memory in different ways. SDL has a numerous of different types, and their scope can be either global or local. A local SDL variable is comparable to a temporary PLEX-C variable. There are also differences between their timer variables (a variable that periodically is increased).

Typing
SDL is a hard typed language, i.e. assignments and comparisons of two variables must have the exact same type, or must be converted so the types on both sides the equal sign respectively the assign sign are the same. In PLEX-C there is no typing at all. 16 bits structured variables can be assigned 16 bits pure and unstructured variables, even eight bits variables can be both compared with and assigned 16 bits variables' values and vice verca. This is not possible in SDL.

Object orientation
One benefit with SDL is that it is object oriented. This benefit is no argument for changing programming language from PLEX-C to SDL since PLEX-C is not object oriented, and this feature cannot be used.

System overview
The SDL environment consist of several tools, e.g. SDL editor, MSC editor, Simulator, Analyzator, log window, code generators, etc. which are integrated and connected to each other within an Organizer. The close relationships between the tools makes the system overview clearer than in PLEX-C development. In the PLEX-C environment, similar tools are used in development, but the close integration and connections between them that SDLtool supports does not exist. This is no disadvantage for a programming language

change though, rather a benefit that not affects the difficulties for converting from PLEX-C to SDL.

Start/restart, forlopp handling and size alteration
Ericsson has some functionality that handles special cases that can occur in the system. The *start/restart* function is used when the system is started or restarted after a fault has occurred in the system. *Forlopp handling* is a function that kills just that process that has stopped for any reason. The benefit is that all other processes can continue as before. Each process has its own database over connected MS's with data that is relevant for the process. Sometimes the size of these databases needs to be changed and it is the *size alteration* function that handles that. These functions cannot be found in SDL, but PLEX-C has them.

Abstraction level
As mentioned before, SDL has a higher abstraction level than PLEX-C. In this case the differences in abstraction level is mostly noticed in the presentation of the source code, i.e. the flow through the program is more obvious in SDL than in PLEX-C. But it is also noticed in the way the different programming languages presents a system in. PLEX-C is not a multitasking programming language (a method to make several programs run parallel), i.e. only one part of the code can be executed at a time and the multitasking has to be considered by the designer, or with other words, the designer decides when his program shall release the processor so other programs can execute. A program in this case is one block. At SDL level, processes (the corresponding to programs) runs parallel, i.e. multitasking. Every block has one *block process* that handles common functionality for the block, and may also have numerous of *individual processes*, and each of them handles a single subscriber in the system (individual). In PLEX-C the subscribers' data are stored in a large database that is constructed like a record in Pascal, or a structure in C. Actually the record, or structure is a large two dimensional matrix were each row corresponds to a certain individual. The individuals are reached with a pointer that says what individual should be handled.

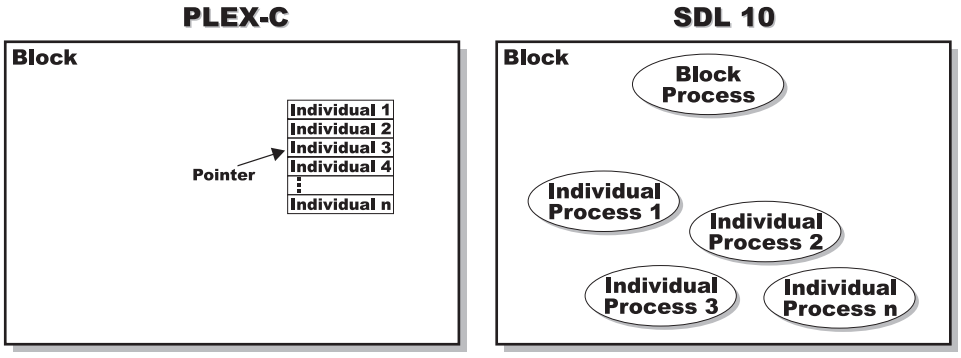


Figure 10. Individuals in PLEX-C and SDL

6.3 Conclusions for similarities and differences

The differences cannot be so extensive that it is impossible to convert from PLEX-C to SDL and the similarities must overcome the differences. The realtime similarities are very good and makes it possible to do the language change, but the abstraction level differences makes it hard, actually very hard. The designers have to reassess and attack the problems in a different way.

7

Reverse PLEX-C code

Previously in this thesis references to a *reverse tool* has been presented, but no deeper explanation about it was provided. The history of the reverse tool's development can be found in section 3.6 and since it still is under development, new versions are released continuously. The tool is developed by Telelogic and integrated inside SDLtool as a menu choice.

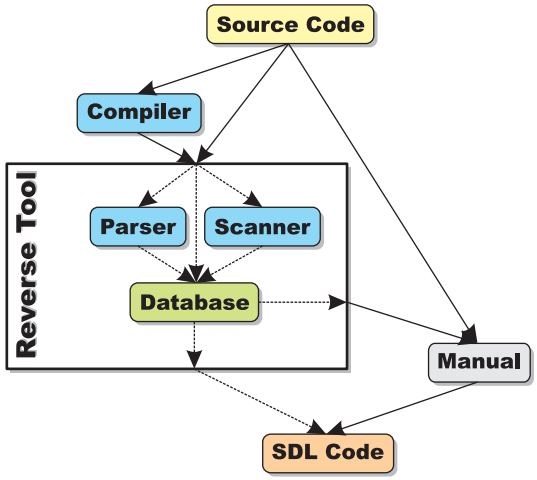


Figure 11. The reverse tool within the reverse process

Figure 11 shows where in the reverse engineering process, described by figure 3, the reverse tool is. Dashed lines are work made by the reverse tool and straight lines are manual work. The reverse tool is mostly interested in the compiled source code, and the source

code itself is only used to achieve comments in the code. The tool produces two things, the uncompleted SDL code, and a log that describes what was not converted and has to be done manually.

As described previously, the reverse tool cannot convert all code automatically. There are two reasons for this, first the tool does not support all PLEX-C constructs, for example linked lists, and second some parts are generated by the SDL2PLEX generator and does not need to be implemented in SDL. The second part is recognized as automatic generated code and more about that below.

7.1 Block division

When converting blocks with the reverse tool, the percentage PLEX statements converted may vary much. The reason is that the tool does not support all constructs, and in different blocks different constructs are used. Comparing a block's construct with its purpose shows a strong connection, and for that reason blocks can be divided into four categories depending of their purpose. The categories were briefly discussed in section 3.6.6 and also in [I5].

Traffic blocks
These blocks handles traffic related jobs, for example handovers and assignments. The percentage of automatic converted statements is high (75% - 95%) (result from [I5] and own experience), since they are the easiest blocks to convert.

Command blocks
An exchange operator can from a terminal adjust the exchange's behaviour by commands. These commands are handled by *command blocks*. These blocks are, compared to other kinds of blocks, relatively small.

Database blocks
These blocks works as databases and stores information, for example about subscribers, cells etc.

Message handler blocks
The nodes in an GSM network, i.e. BSC, MSC, BTS, etc., are communicating with standardized messages. *Message handler blocks* converts data retrieved within these messages to the data format used internally in the node.

The reason for traffic blocks being the most successful blocks to convert is that the reverse tool is mostly tested on these kinds of blocks, and for that reason most of the trouble reports to the developer is from converting such blocks.

7.2 Reverse tool unsupport

Since the reverse tool is under development, new versions with more support for uncovered constructs will be implemented in the future. The purpose of this section is to show

the reverse tool's support and unsupport at the time when our work were made. There can exist more unsupported constructs not described here though. The reason for this is that we had not the ability to convert sufficient number of blocks, or different kinds of blocks.

As described in section 3.6.6, the success for converting blocks depends on the kind of the block. The reason is that the reverse tool supports some constructs more or less. For example the reason for low converting percentage when converting command blocks derives from how the state is set in the PLEX code. This differ between command blocks and traffic blocks.

Other general constructs not covered by the reverse tool are:

- **Linked lists** - will be supported in later versions
- **Several starting points within a process** - also supported in later versions.
- **Forlopp statements inside the individual process** - instead of the right subroutine call, a comment with the untreated linenumber in the PLEX code, is put in the SDL code.
- **Timers** - hard to convert correct, sometimes the reverse tool recognizes timers and sometimes not.
- **Individual pointer** - In SDL10 half the PId value corresponds to the individual pointer in PLEX-C. Converting between these may sometimes be difficult for the reverse tool.

7.3 Automatic generated code

Some of the statements that not are converted, are automatic generated and need not to be converted into SDL code, but to get the automatic generated code correct, directives to the PLEX generator must be set. There exists scripts that helps the designer to find out these directives. Automatic generated code concerns:

- **Size alteration** - change number of individual processes, or change the size of a vector during execution.
- **Scanning individuals** - a procedure to find out if an individual has got any time out in their timers.
- **Forlopp** - statements for forlopp initiation.
- **Start / Restart** - when the system is started / restarted some general procedures are run.

7.4 Time estimation

It is important to have knowledge about how long time a conversion will take since the project leaders must plan how much effort every conversion needs, i.e. staff, costs, time, etc. The KomPlex project [I5] presents formulas for estimating conversion times, but since

this is confidential information and intended to be used by Ericsson personnel only, we cannot present the formulas or information about them here.

7.5 Reverse tool in our work

We have converted two blocks inside the BSC from PLEX-C to SDL10 (for further information about BSC, see appendix A). The first block can be classified as a traffic block and the second as a message handler block. Both blocks have about 1100 PLEX-C statements each. The first block, `RMASS`, handles a part of the function *assignment in serving cell*, and the second block, `RMHAIUL`, handles the communication from the CPR blockgroup to its environment. The conversion of these blocks were successful due to the high percentage converted number, in fact almost all code that could not be automatically generated were converted. The percentage converted number will never be 100% because the automatic generated code will never be converted, and such code exists in all blocks. Even if we ignores that, the converted code (the SDL code) is not fully correct. The code and the type definitions are converted, so are the signals, but not the types on signal parameters. They have to be corrected, irrespective of the percentage converted figure. There may also occur other problems due to unsupported constructs by the reverse tool.

Reverse Data

Block	Type	% coverage	# statements	# signals	# states
RMASS	Traffic	89,0 %	1162	79	14
RMHAIUL	Message Handler	90,9 %	1106	80	3

RMASS
The unconverted statements were all related to automatic generated code (implemented with directives in SDL), except for some forlopp subroutine calls. Problem one was discovered when converting this block.

RMHAIUL
As in RMASS, the unconverted statements were all related to automatic code, except for some forlopp subroutine calls. Problem two and three were discovered when converting this block. When converting this block, we also made time estimations according to the formulas presented by the KomPlex project [I5]. The estimated total conversion time became 43.1 hours, and the actual conversion time became 42.8 hours. This may look like manipulated figures, but they are not. Remember though, that the KomPlex project [I5] also has tested the formulas with good results, so they can be seen as acceptable.

7.5.1 Problems

Even if the reverse tool converted 90% of the source code, we had three major problems in the conversions. The first problem regards abstraction level differences, the second typing

differences between SDL10 and PLEX-C, and the third regarded variables used by several processes. The reverse tool should know about these problems and reconstruct these parts.

Problem 1
Figure 10 describes the difference in how individuals are represented in PLEX-C and SDL10. It is these differences that make our first problem. In many blocks inside block-group CPR, individuals are mapped, i.e. same individuals has the same value on its individual pointer in different blocks. This is a benefit since the individuals does not have to be allocated in all blocks. Instead all individuals are allocated in one block and then mapped to the same place in all other blocks. This problem is common within almost all traffic blocks. It is recognized when the initiation signal is sent to an individual process instead of to the block process, where the allocation would occur and a new individual process should be created. This means that no individuals can be created in runtime.

Solution 1.1
One solution could be sending the initiation signal to the blockprocess and there create a new individual process. But one can never decide what PId value a created process should have, and for that reason the mapping between blocks will be corrupt. Instead *all* signals have to be sent to the blockprocess and there mapped to correct individuals, which means that all input signals are duplicated as internal signals. This is not a good solution. Figure 12 shows how two signals, signal 1 and signal 4, are sent to the block process. The blockprocess uses the map table to find what individual the signals are intended for, and bypasses the signals to correct process. In this case the external individuals (towards this block) 1 and 4 are mapped as internal individuals 3 respectively 2.

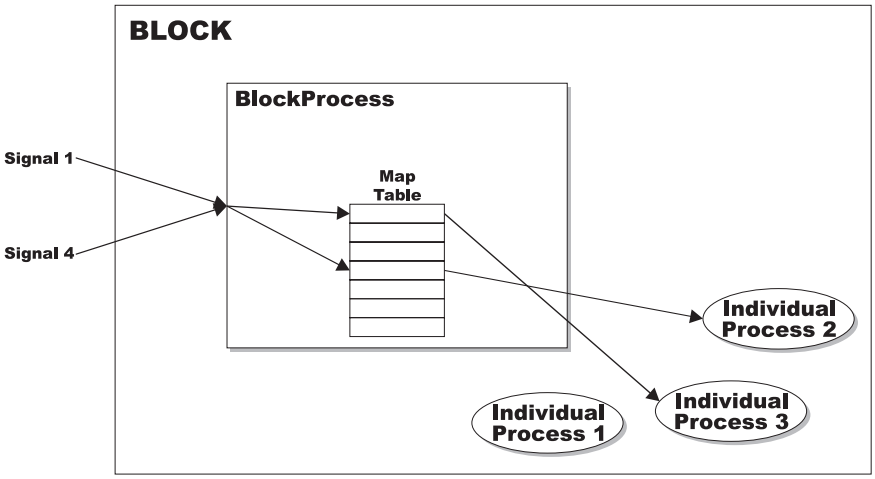


Figure 12. Signal mapping

Solution 1.2
If one could create processes with predefined PId values, the problem should be solved directly. This solution is not a pure code solution, instead Telelogic has to redesign Tau to include this feature. That is more a political problem and probably not easy to fulfil.

Solution 1.3
Individuals that are idle (in state IDLE in PLEX-C) should not exists in SDL10, but be created when an initiation signal is received. It is the block-process that receives this signal and creates a new individual process. Later when the individual is finished with its work and would go idle in PLEX-C, the individual process in SDL10 is terminated. This is how it is meant to implement blocks in SDL10. Suppose instead that all individual processes exists all time and goes to an idle state when idle instead of terminating. This means that the initiation signal can be sent to the individual process that corresponds to the mapped individual that sent the signal, and the mapped structure between blocks is taken care of and this solution may solve this problem. We used this solution in our work with good results.

Problem 2
The second problem is about typing differences. In PLEX-C a variable is seen as a set of bits. The variables can be structured or not. Assignments and comparisons between structured and unstructured variables are permitted, and also assignments/comparisons between variables of different sizes, e.g. eight bits variable compared with a 16 bits variable, are permitted. The easy typing has both benefits and disadvantages. One disadvantage is that variables with different structures can be assigned to or tested against each other without warnings and error messages when compiling. But this can be a benefit too since variables with different structures may have the same meaning, and that is very common in PLEX-C. In SDL10, on the other hand, typing is strong, very strong. Variables that are assigned or compared with each other must be declared as the exact same type. This is a problem when converting code from PLEX-C to SDL10.

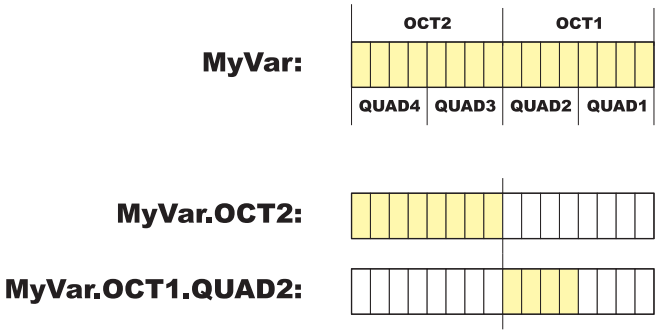


Figure 13. Variable structures

Variables in PLEX-C are often structured. The most common size is 16 bits. This word (16 bits) is divided into two bytes called `OCT1` and `OCT2`. These bytes are also divided into two pieces each, `QUAD1` and `QUAD2` respectively `QUAD3` and `QUAD4`. The `QUAD` variables (the divided structures can be seen as single variables too) has the size of four bits. Sometimes even a smaller division is made. That is similar to the quad division, but not handled here. A structured variable can be accessed at different levels, as 16 bits, eight bits, or four bits. It is important that substructures, or variables within the same structure not are named

equal since that is not permitted in PLEX-C. This could for example be `QUAD3` and `QUAD4` named as `QUAD1` and `QUAD2` in the structure above (figure 13).

In PLEX-C, assignments between variables of different sizes are permitted too. The rules that controls this is shown in figure 14. This should never be used in design though, since the possibility for misunderstanding. In figure 14 both variables are declared as 16 bits structures as described in figure 13.

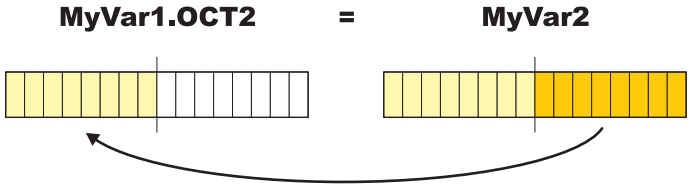


Figure 14. Variable assignments with different sized variables

This “type problem” is not only local to the block, but also global. Let us say that you have converted two or more blocks and wants to simulate their behaviour in Tau. Then you have to connect the blocks to each other with predefined signals. The signals are converted together with the block that uses the signals, and if two blocks uses the same signal (they do if they communicate with each other), two different declarations of that signal will occur. The differences between them will be the type declarations of the signal parameters.

Solution 2
Unfortunately the reverse tool is not much help. Instead it might even make it worse. The problem for the reverse tool is to identify the structure of variables in the code, and if there is a slightly difference, new structures at SDL-level is produced. This is a problem when comparing and assigning variables in SDL, which will theoretical have the same structure, but are declared as different types. Instead, we wrote our own structures to the latter converted block (this block has lots of structures and was hardest to convert types within). Also operators to convert between a structure and integer, were produced (written in C++).

In conclusion, the solution to this problem is quite easy. When signals are converted the structures used in the signal parameters are also converted. These structures are later used to declare most of the variables in the converted block. The variables that cannot be declared as these types retains the type declaration that the reverse tool set. This does not only solve type conflicts with signal parameters, but also the problem that the reverse tool has when converting structures.

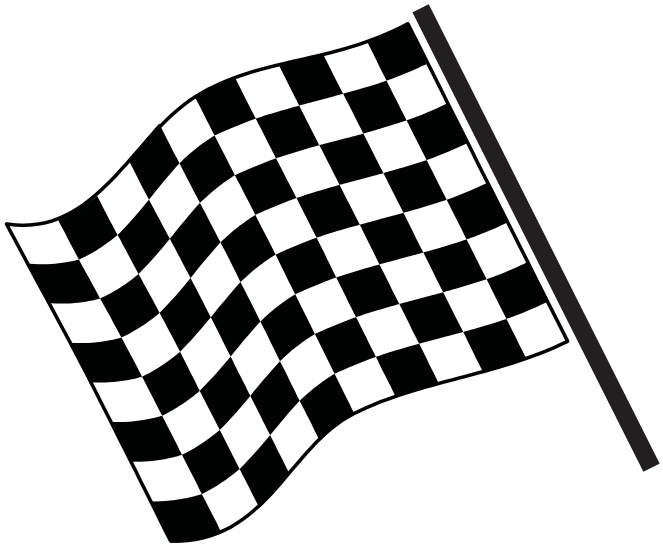
Problem 3
In PLEX-C variables can be declared differently.

- **Temporary -** These variables loses their value when getting out of scope, i.e. when changing state, calling subroutines, etc.

- **Common store -** These variables remember their value even if the state is changed or a subroutine is called. They are technically declared outside both the blockprocess and the individual process.
- **In records -** Variables belonging to a certain individual are stored in a record (a kind of a vector). These variables corresponds to variables described in the individual process.

In some blocks the common stored variables are accessed and set by both the blockprocess and the individual process. The reverse tool solves this by declaring the variables in both process types, and for that reason one variable at PLEX-C level becomes two different variables with absolutely no connection between each other at all. This is not correct, since in the PLEX-C source code just one variable is used by both processes.

Solution 3
In SDL, variables must be declared within a process and they are not accessible outside that process [P5]. One can declare a variable as *revealed* though, which makes the variable readable for other processes, but they cannot store any value in it. To store values in other processes variables a *remote procedure* must be implemented. Remote procedures are implemented in the blockprocess and can be accessed by other processes. By applying the value to store as an inparameter to the remote procedure, it can store the value to the local variable in the blockprocess.



8

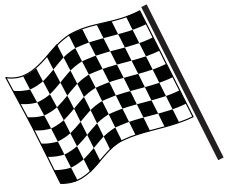
Conclusions

Our work shows that it is possible to convert blocks inside the BSC from PLEX-C to SDL10, with assistance from the reverse tool. The tool reduces the conversion time towards manual conversion with approximately 40% [I5]. Problems that occur when converting source code have mostly its origin within the differences between the two programming languages, and can seldom be derived from either the reverse tool or the SDL2PLEX generator. The greatest differences that we have discovered within our work were about typing level (strong or easy), representation-, and variable- differences.

The first difference convey in problems for converting certain assignments and comparisons from PLEX-C to SDL10. PLEX-C is easy typing and SDL10 is strong, which means that some assignments / comparisons stated in PLEX-C code are not allowed in SDL10. The representation difference derives from how individuals are represented in the two programming languages. Sometimes individuals are “mapped” in PLEX-C, but in SDL10 the individuals should be “created”. When creating a process in SDL10 one cannot decide what reference number it should have, which theoretical is done in PLEX-C. The last difference is about variables. In PLEX-C *common stored* variables, i.e. variables declared outside all “processes”, can be used. In SDL10 variables must be declared inside a process and belong to such one.

Unfortunately the revers tool does not support these differences and cannot convert these parts correctly, or even not convert them at all. With some manual effort, as adding and correcting to the converted block (described in appendix C), its functionality will be the same in both languages.

Some blocks are easier than others to convert, i.e. the reverse tool converts larger parts correct and SDLtool and the SDL2PLEX generator supports some PLEX-C functionality



more than others. Since the reverse tool is mostly tested on traffic blocks, these are the best supported blocks from both the reverse tool, SDLtool, and the SDL2PLEX generator. But with further adjustments to these three products, they will support conversion of the other block categories as well as for traffic blocks.

KomPlex [I5] presented formulas for time estimations when converting a block. They have been tested once, and since no other tests have been made we cannot say whether they are good formulas, or not. Since the type of block to convert affects how much the reverse tool converts, this should be a factor in the first formula.

Our work has also resulted in a process for how to convert a block. The script has only been tested once though, but the intention is to update the script each time it has been used.

We consider our work as successful, and recommend usage of the reverse tool. The problems that still exist (not have been found yet) will probably be easy to solve. With more usage of the tools (reverse tool, SDLtool, and SDL2PLEX generator), they will gain more quality.

8.1 Differences

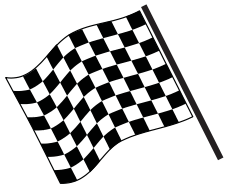
Our work is unique in the way that no other project with same intentions that we had has been launched at Ericsson before. KomPlex [I5] had similar intentions, but focused on the connections and interactions between blocks at SDL-level. If a block was hard to convert, they skipped it and took another block. In our work, we have looked for problems when converting blocks, and tried to find solutions.

Blocks have been converted before though, but either without the reverse tool (manually) [I6], or with different intentions.

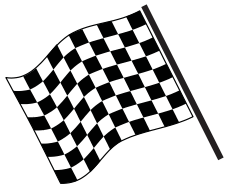
8.2 Future work

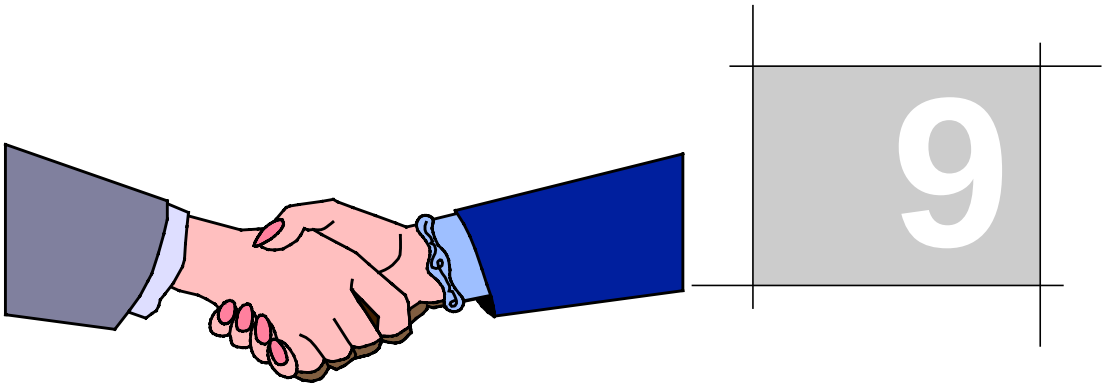
Our work has just been concentrated on reverse engineering PLEX-C code to SDL10 code. The reverse tool aiding the conversion has some lacks that must be provided to a conversion project manually. By using the reverse tool and thereby finding these lacks and report them to the developing company Telelogic, the tool will gain quality and manage to make better conversions considered efficiency, correctness and coverage. Most of the tests on the reverse tool have been done with traffical blocks. By testing other kinds of blocks more knowledge of differences between the two languages will be gained. This knowledge can be used to apply solutions to the problems that the differences make.

When developing products of the size which Ericsson does, it is important to have well defined processes for the development. Ericsson has that. Since SDL and PLEX-C are not at the same level of abstraction, Ericsson needs to make adjustments to their processes if they change their programming language. Processes for source code conversions must be developed too. Similar tools to the ones used today in the development process exist in



Tau, but Ericsson does not have enough knowledge about all benefits and how to use the tools in the best way.





Acknowledgements

First of all I would like to thank my family, Maria Holmqvist and Casper Berg, for all support.

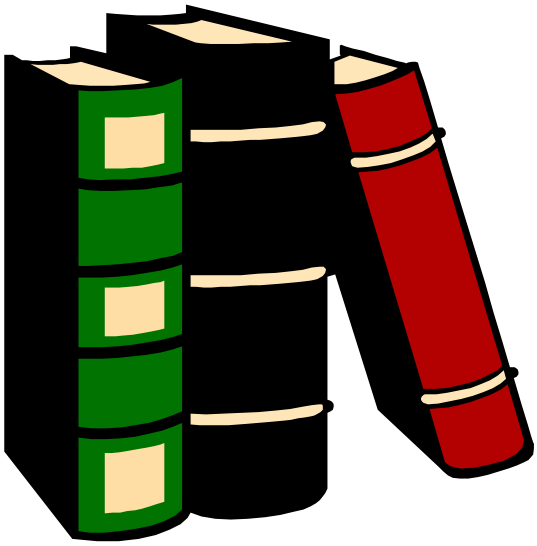
I would also like to thank Anders Dellien, Magnus Persson and Dirk Auchter at Telelogic, who has helped me with my SDL systems when I got stuck, and Chris Verhoef at the University of Amsterdam who has helped me with reverse engineering research.

Huge thanks to my tutors, Magnus C. Ohlsson at the Department of Communication Systems, Lund, and Jörgen Palm and Henrik Cosmo at Ericsson Radio Systems AB for making this MS thesis possible to manage.

Lots of thanks to the people who has read and complained at my work and helped me discussing different kinds of troubles, Tommy Nordgren, Tom Nilsson, Linh Trang and Markus Berg.

And last I would like to thank the radio show *Pippi Rull*, which has made my work effort to slow down almost every monday to thursday 3 pm to 4 pm.

Thanks all!



10

References

10.1 Public Resources [Px]

[P1] Ericsson Telecom AB, Telia AB
"Att förstå telekommunikation 2"
Studentlitteratur, Lund 1998
ISBN 91-44-37811-4

[P2] Telelogic AB
"Introduction to SDL and SDT"
rev 3.2, 1997

[P3] Mark van den Brand, Paul Klint, Chris Verhoef
"Core Technologies for System Renovation"
Technical report, University of Amsterdam.
Available at: `http://adam.wins.uva.nl/~x/reverse.html`

[P4] E.J. Chikofsky and J.H. Cross
"Reverse engineering and design recovery: A taxonomy"
IEEE Software, 7(1):13-17, 1990

[P5] Ferenc Belina, Dieter Hogrefe, Amardeo Sarma
"SDL with applications from protocol specefication"
Prentice Hall, Great Britain 1991, ISBN 0-13-785890-6

[P6] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Mueller, J. Mylopoulos
"Code migration through transformations: An experience report"
Technical report, University of Waterloo Dept. of Electrical Eng.

[P7] Peter Aiken, Ojelanki K. Ngwenyama, Lewis Broome
"Reverse-Engineering New Systems for Smooth Implementation"
IEEE Software, March / April 1999

[P8] Peter Aiken
"Data Revere Engineering: Slaying the legacy dragon"
chapter ***" The necessity of Data Reverse Engineering"*** written by E.J. Chikofsky
McGraw-Hill Companies, USA 1995
ISBN: 0-07-000748-9

[P9] Spencer Rugaber
"Program comprehension"
Technical report, Georgia Institute of Technology, 1995
Available at: `http://www.cc.gatech.edu/reverse/repository/encyc.ps`

[P10] Jean-Marc DeBaud, Spencer Rugaber
"A software Re-engineering method using domain models"
Technical report, Georgia Institute of Technology
Available at: `http://www.cc.gatech.edu/reverse/repository/domain-based-RE.ps`

[P11] John J. Marciniak (Editor in chief),
"Encyclopedia of Software Engineering",
USA, 1994,
ISBN: 0471-54004-8

[P12] M.G. Rekoff,
"On Reverse Engineering",
IEEE Transactions on Systems Man and Cybernetics, SMC-15(2), 1985

10.2 Internet Sites [Wx]

[W1] Web address: **`www.telelogic.se/solution/language/sdl.asp`**
Telelogic AB

[W2] Web address: **`www.webproforum.com/telelogic1/index.html`**
Telelogic AB

10.3 Internal Ericsson documents [Ix]

[I1] Ericsson Radio Systems AB
"CME 20 System Survey Training Document EN/LZT 120 226 R5B"
Stockholm 1996

[I2] Lecture by Birgitta Strandberg, Ericsson Radio Systems AB
Hässleholm May 13, 1999

[I3] Ericsson Telecom AB
"PLEX-C1"
Stockholm 1996

[I4] Conny Johansson
"Root cause analysis of SDL Reverse"
Ericsson document number 1/0363-4/FCP 105 9017 Rev A

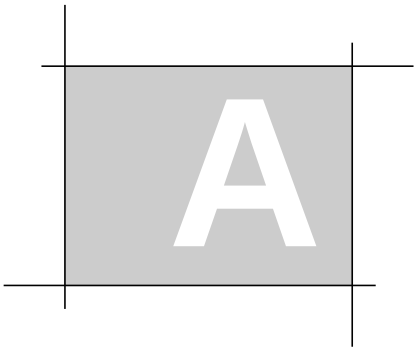
[I5] Barbara Reisner
"Final report for: SDL Reverse prototype (KomPlex)"
Ericsson document number 0363-FCPW 101 34 Rev A

[I6] Tomas Kostenius
"Final report for: SPOT (SDL10 PilOT)"
Ericsson document number EPK/DG-98:051 Rev A

[I7] Stefan Persson
"Problem med för många FORLOPPs-releaser i APZEmu'n"
Ericsson document number EPK/DX - 98:080

[I8] Telelogic AB
"Plex to SDL 1.0 User's Manual"
Id: SMO99-XPR-20 version 1.1

[I9] Anna Wetekam
"MSC / SDL10 Layout Guidelines"
Ericsson document number 8/000 21-FCK 114 2004 Uen Rev A



GSM / BSC

The GSM system is large and no deeper explanation of the system is done here, except for one part, the BSC (Base Station Controller). The reason for this is that it is on blocks, or small parts inside the BSC that reversed engineering will be applied.

A.1 History

GSM (Global System for Mobile communication) was “created” in 1982, as a proposal to specify a common European telecommunication system. It took many years to decide what techniques that should be used, for example should it be a digital or an analog system, and what access method should be used. Not until July 1, 1991 complete GSM systems were running all over Europe. All the planning and discussing conducted in that many operators started at the same time and a very large potential market could open. That is one reason for the popularity of the system and the fact that the number of subscribers has grown fast (and still does).

Since the start the GSM standard and all existing systems have been under development. New services, both for operators and subscribers have been implemented. Also, the numbers of subscribers are increasing which affects the load on the system, and therefore the system needs to be upgraded.

A.2 Techniques and restrictions

GSM is a digital mobile phone system, that uses the TDMA (Time Division Multiple Access) technique as access method. The TDMA technique has several advantages towards the broadband alternative FDMA (Frequency Division Multiple Access). It splits

the time into several time slots (GSM has 8 time slots) and uses each time slot as one channel, which means several channels per frequency. In FDMA each channel get its own frequency, but instead the bandwidth is higher. Many channels, that TDMA gives, were chosen instead of the high bandwidth (compared to TDMA) in FDMA (see figure 15). The time slots in TDMA are separated by a little gap called *guard period*, to avoid that the time slots will overlap each other. This is necessary because that the subscribers are moving during transmission and the fact that the signals take a little time to move from the subscriber to the base station [P1].

The low bandwidth that TDMA gives trespasses on the data transmission rate and on the sound quality. A technique that is based on how the human speech organ is build solves the sound problem and reduces the need of a high band width. If a subscriber wants a higher data communication rate, several time slots can be connected to the same subscriber, i.e. mobile station (MS), but both the operator and the MS must support this feature. It will also cost more for the subscriber than a usual phone call [P1].

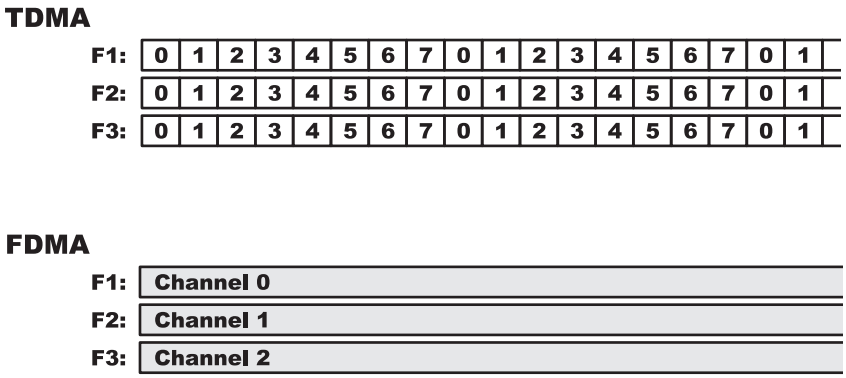


Figure 15. Differences between TDMA and FDMA

The bandwidth in an ordinary stationary phone is 64 kbit/sec, and that is only for the speech. In a GSM phone only 13 kbit/sec are available, and here must signalling be included. When transferring data from and to an MS, the highest bandwidth is 9.6 kbit/sec [P1].

A.3 Structure

The overhead design of GSM is divided into two major parts, the Switching System (SS) and the Base Station System (BSS). These parts contain each several units that have its own purpose. Figure 16 shows the structure of CME 20, Ericsson's implementation of the GSM standard [I1].

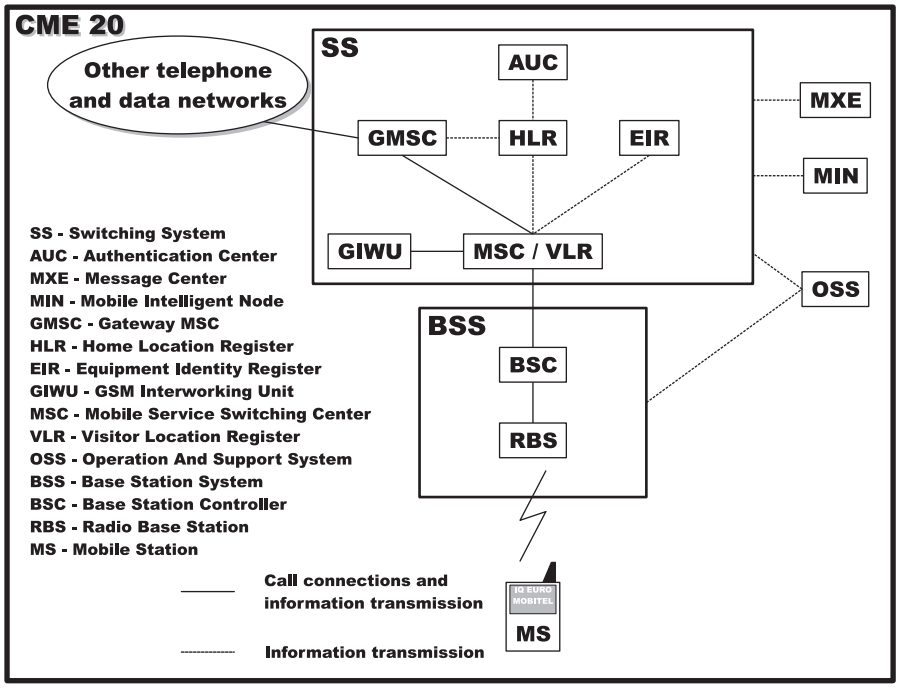


Figure 16. CME 20 system [I1]

As one can see, there are many parts in the CME 20 system. For our purpose, only the BSS part, and particular the BSC, is of any interest, because it is deep inside the BSC where the code to be converted (with reverse engineering) can be found.

In simple terms the SS part connects an MS to either another MS, or terminal, in the same net, or another phone (stationary or cellular), terminal or server, in another net.

The units inside the SS part that are connected with dashed lines are servers that holds information about which MS's that are connected to the system right now, and where they are. There is also a database in which the system can check if a certain MS are not stolen or in another way not permissible to use the network [I1].

The OSS unit contains functions to overview the network, and the MXE unit contains functions for SMS services, voice and fax mail, and cell broadcast. The MIN unit handles the intelligent network services in CME 20 [I1].

A.4 BSC

The BSC is located in the BSS part. Its purpose is to handle all the radio-based functions in the system. The BSC controls underlying RBS's (Radio Base Station, called BTS in the GSM standard) and the BSC in CME 20 can handle many RBS's. This feature reduces the traffic between BSC's and MSC's, but it brings a need of a powerful and complex BSC [I1].

A BSC makes lots of things. Over a million of calls per day are handled and distributed by a single BSC. The basic functions, that a BSC has, are [I1]:

- Radio network management
- Radio network performance monitoring
- Operation, maintenance and administration of RBS
- Speech coding and rate adaptation
- Transmission management towards RBS
- Handling of the radio resources during MS connection

The BSC itself is divided into subsystems and these are shown in figure 17.

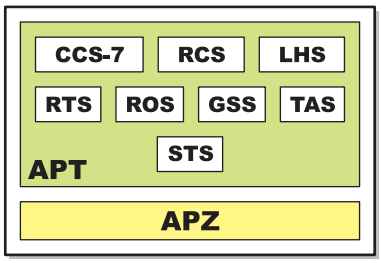


Figure 17. Subsystems inside BSC

Some of the subsystems are divided into blockgroups, and in the subsystem RCS we have a blockgroup called CPR. It is inside this blockgroup where the blocks to be converted can be found.

Figure 18 will show in another way where the interesting blocks can be found in the CME 20 system.

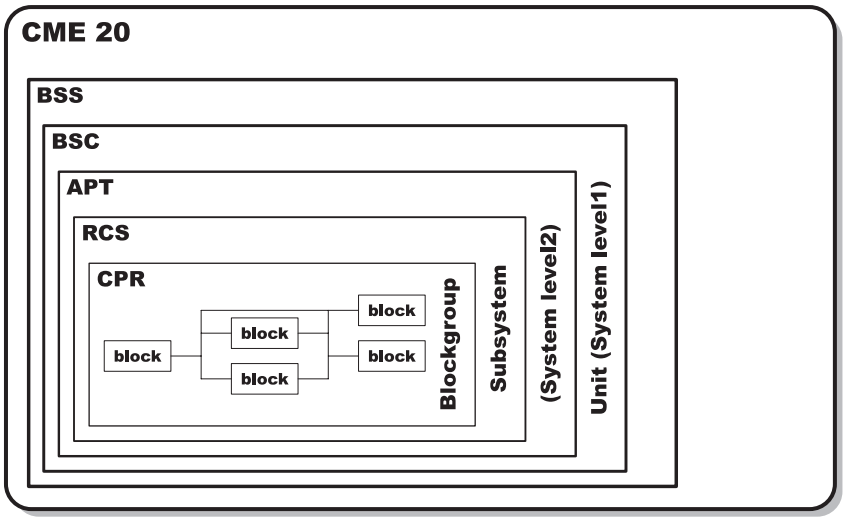


Figure 18. Block hierarchy inside CME 20

The functions handled by these blocks are among others paging, signalling connection setup, assignment, handover, resource level supervision, cipher mode control, classmark distribution, transfer of BSS transparent messages, short message service (SMS), connection release and traffic event measurement in radio network [I2].

A.5 Function explanations

The functions were explained by [I2].

A.5.1 Paging

In the stationary telephone network all telephones, or terminals, are connected to a fixed point. The network knows all the time where this specific terminal is. If we want to move the terminal, the connection point will be changed, but this rarely happens during a phone call. In a cordless network, like GSM, there is no fixed connection point for an MS. When someone calls an MS the network has to ask the specific phone where it is. This is called paging. Paging is costfull, because if the network has no idea were the MS is located, it has to page its hole supplying area. This would break down the network in a few seconds. Instead all MS's has to register themselves when they are turned on or when they change location area. A location area consists of a number of cells. They can be under the same BSC or under different BSC's, but they have to be under the same MSC. The MS must also deregister when it is turned off, so the paging procedure does not load the net too much [P1].

The paging function is used when the GMSC wants to know exactly where a specific MS is located, for example when someone from another network is calling the MS. But since

the MS is located in a location area, the paging is done just there, and only if the network is not highly loaded [P1].

A.5.2 Handover

Handling handovers is a major part in the BSC. The handover concept means that an MS change its communication channel to the system in some way. This is done to maintain good radio transmission quality, save calls from being disconnected or blocked, or regulate the load on the system. Handovers can be within the same cell or between two cells [I2]. A cell is a small geographical area with a set of frequencies [P1]. One antenna often represents one cell.

Three different cases of handovers can occur [I2].

1. Intra BSC, Inter Cell handover. Handover between two cells controlled by the same BSC.
2. Inter BSC handover. Handover between two cells controlled by two different BSC:s.
3. Intra BSC, Intra cell handover. Handover inside the same cell.

The last item can be little strange. Why making a handover inside a cell? Suppose that the frequency that the communication are held over, are reflected (by buildings etc.) in a strange way, and the receiving condition gets poor. By changing frequency (and also time slot) the receiving condition can be improved.

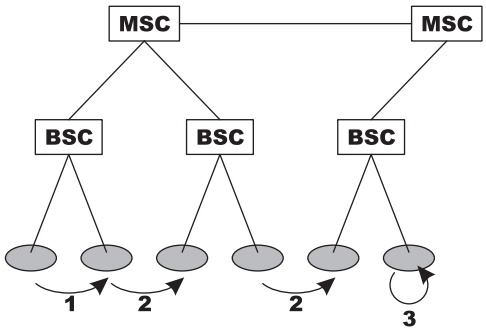


Figure 19. Different handover cases

Figure 19 shows the three handover cases. The second case can occur in many ways, both under the same MSC and under different MSC's. It is the BSC who decides whether a handover will be done or not, based on, for example receiving conditions [I1].

A.5.3 Signalling connection setup

When a connection is going to be established between an MS and the GSM net, signalling is done over a common channel. To release this channel to other new connections, a new

channel has to be allocated. This function handles the transfer from a common signalling channel to an allocated channel.

A.5.4 Assignment

To set up a phone call much signalling is needed in the beginning. The phone number is one small part of all the data that has to be exchanged before the phone-call is totally set up so the subscriber can start to talk. When all signalling is done, the connection will be transferred from a signalling channel to a traffic channel and that is done by the assignment function.

A.5.5 Resource level supervision

If some connections require more than one time slot (e.g. data transfer), but not enough time slots are available, this function tries continuously to upgrade these connections to the required amount of time slots, i.e. when a time slot is released from another connection, it is allocated by this connection.

A.5.6 Cipher mode control

Data sent between an MS and the GSM net is always encrypted. This function selects and sets an encryption alternative that is both permitted by the MSC and supported by the MS.

A.5.7 Classmark distribution

This function retrieves information from the MS about its capabilities. It can be RF power, ciphering algorithms and multislot class.

A.5.8 Transfer of BSS transparent messages

With this function data is sent transparent through the BSS, i.e. a kind of direct contact between the MS and the MSC. This is used for instance to send the dial tone or noise. The noise is sent in convenient purpose. When the person that the subscriber of the MS is talking to is quiet, no sound is sent to the MS, but to ensure the MS subscriber that the connection still exists, noise is sent to the MS.

A.5.9 Short message service (SMS)

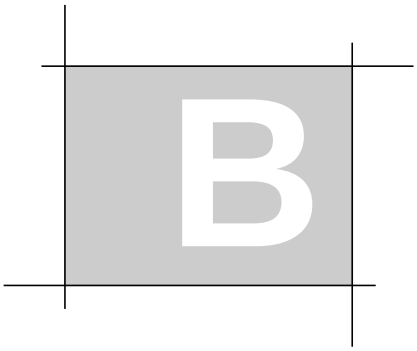
Short messages are text messages (up to 160 characters) that can be sent from and received by an MS. A special server, the MXE unit, handles and distributes these messages and they are sent transparently between the MS and the MXE.

A.5.10 Connection release

A connection can end in many ways, and can happen for instance when a phone-call is ended, the MS is handovered to an BSC external cell, hardware faults occurs or the MS is considered lost (too far away from any antenna). This functions handles this and releases all occupied resources e.g. channels.

A.5.11 Traffic event measurement in radio network

This function is the statistical part in the system. It counts different events in the system such as assignment attempts, dropped connections and handovers. With help from this data the status of the system can be defined, and if necessary, adjustments can be done.



Abbreviations

Abbreviations

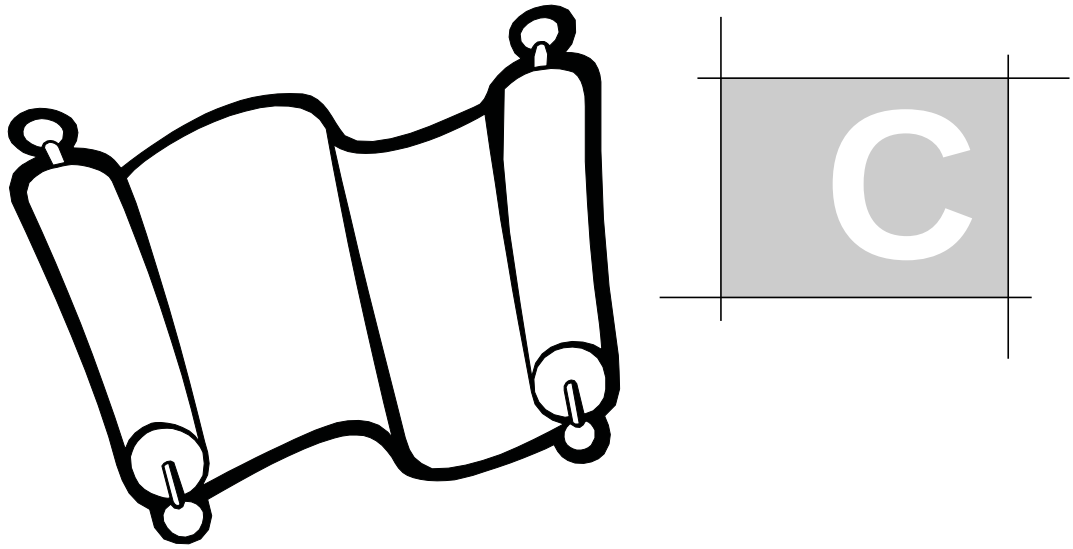
Acronym	Explanation
ADT	Abstract Data Type
ASA	AXE 10 processor assembler language
ASN.1	Abstract Syntax Notation one
AST	Abstract Syntax Tree
AXE	Ericsson’s well famous exchange
BSC	Base Station Controller
BSS	Base Station System
BTS	Base Transceiver Station
CAD	Computer Aided Design
CASE	Computer Aided Software Engineering
CCITT	ITU-T’s old abbreviation
CME 20	Ericsson’s implementation of the GSM standard
CPR	Connection PRocess, a blockgroup inside the RCS subsystem in CME 20
DECT	Digital Enhanced Cordless Telecommunications

Abbreviations

Acronym	Explanation
DOA	Department of Accounts
DP&T	Department of Personnel & Training
EFSM	Extended Finite State Machine
FDMA	Frequency Division Multiple Access
FSM	Finite State Machine
GMSC	Gateway Mobile service Switching Centre
GSM	Global System for Mobile communication
HMSC	High level MSC
HSDL	High level SDL
ITU-T	International Telecommunication Union - Telecommunications Standardization Sector
MIN	Mobile Intelligent Node
MS	Mobile Station
MSC	Message Sequence Chart (in SDL environment)
MSC	Mobile service Switching Centre (in CME 20 System)
MXE	Message Centre
NMT	Nordic Mobile Telephony
OSS	Operation and Support System
PId	Process Id
PL	Parameter List
PL/IX	Programming language developed by IBM
PLEX-C	Programming Language for EXchanges, C-version
PLEX-M	Programming Language for EXchanges, M-version
PLEX2SDL	Reverses PLEX-C code to SDL10 code (same as reverse tool)
RBS	Radio Base Station
RCS	RadioControl System, a subsystem inside the BSC in CME 20
Reverse tool	Reverses PLEX-C code to SDL10 code (same as PLEX2SDL)

Abbreviations

Acronym	Explanation
SD	Signal Description
SDL	Specification and Description Language
SDL10	Adaption of SDL to fit Ericsson’s needs
SDL2PLEX	PLEX-C code generator in Tau (generates PLEX-C code from SDL10 code)
SDLtool	Adaption of SDT to fit Ericsson’s needs
SDT	SDL Design Tool
SMS	Short Message Service
SPI	Source Program Information
SPL	Source Parameter List
SPOT	SDL PilOT
SS	Signal Survey (in PLEX-C documentation)
SS	Switching System (in CME 20 System)
Tau	Telelogic’s development environment for SDL
TDMA	Time Division Multiple Access
TheMAT	The Metadata Access Tool
TTCN	Tree and Tabular Combined Notation



Conversion process

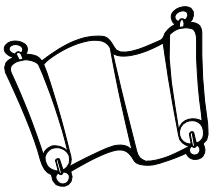
Converting source code from PLEX-C to SDL10 is not difficult, but there are many steps to perform before a complete and validated SDL10 implementation of the origin PLEX-C code exists. This process helps a designer execute a conversion. The reverse tool (PLEX2SDL) is used to do the major part, but it can neither convert all statements in the source code, nor evaluate what parts that should be in different processes.

The process is written on the basis that the reader is familiar with the SDL environment and the differences between PLEX-C and SDL10. It describes “one block at a time” conversion, but if several blocks should be converted, the process should be executed once per block. The process is divided into different phases for different tasks. Each phase is described with an activity flow that shows step by step how to do.

To make the process evaluation phase valuable, it is important to take notes about the process during the conversion.

Some steps that should be included in the process is excluded since they were never executed in our work. These steps handles document handling and some document producing (according to Ericsson’s development process), but they should be similar as within Ericsson’s normal development process.

In the process the *designer* is the person who executes the conversion process, and the *main project* is a project concerning several blocks to be converted.



C.1 Time estimations

The KomPlex project in Aachen [I5] resulted in formulas for calculating the total time for a conversion of a block. Since this is confidential information and intended to be used by Ericsson personnel only, the formulas are not presented here but can be retrieved in [I5].

C.2 Activities

The process is divided into eight phases as figure 20 describes. Each phase is divided into a number of steps, or activities that should be performed. Besides the activities, the purpose, hints, comments and input / output denoted to the phase is described too.

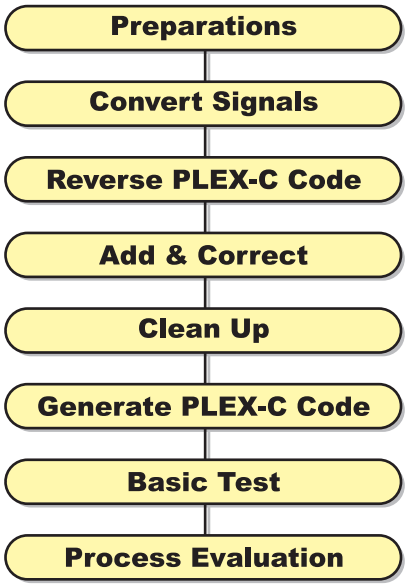
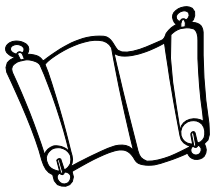


Figure 20. Phases in conversion script



C.2.1 Preparations

Purpose

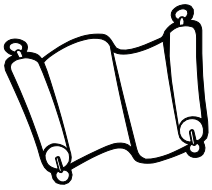
This phase is intended to prepare and setup a single block conversion project.

Input

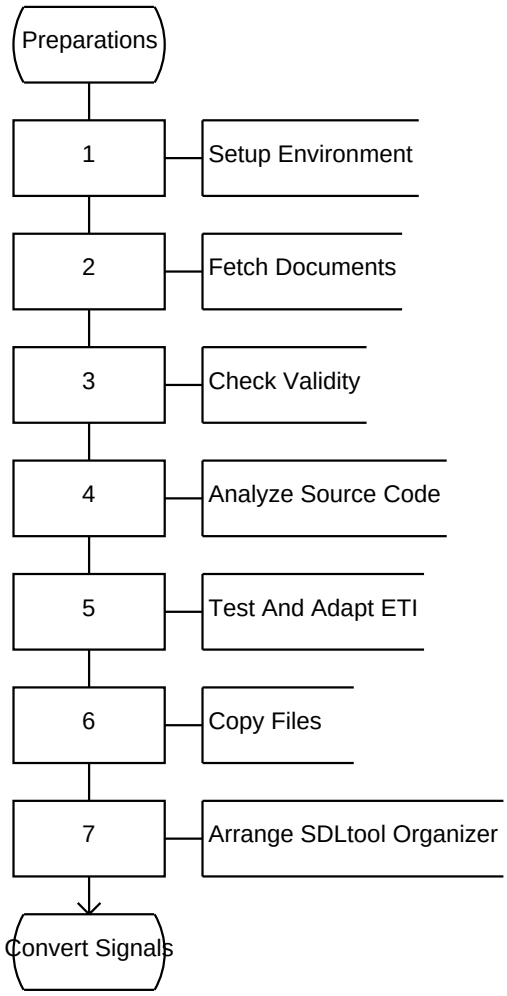
- The block that will be converted.
- File `GeneralRoutines.sun`
- File `PlexTypes.sun`
- File `CommandRoutines.sun`
- File `[blockname].ssurv`
- File `[blockname].param`
- File `[blockname].program`
- File [blockname]_ETI.script (may be named different)

Output

- A UNIX directory structure with necessary documents in place.



Activities

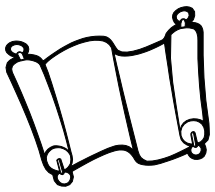


1. Setup environment

A UNIX directory structure must be created. The structure is defined by the main project, but how the structure is built up does not affect this process except for the following exceptions:

- An empty directory, `signals`, must exist in the directory where the SDLtool associated files are stored, e.g. `*.sdt` files.
- The fetched files (described below) and their analyzed outcome must be stored in the same directory.

The directory structure shown in figure 21 is a proposal, and we refer to this structure in this process. Note that the structure is for one block only, and each new block that will be converted must have its own directory structure.



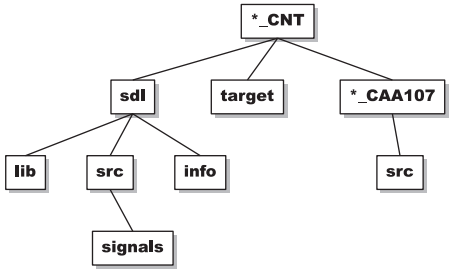


Figure 21. Directory structure for a reverse project

The asterisk (*) should be replaced with the blockname. When the directory structure is created, three files must be copied into it. They are stored in the SDLtool installation directory, but must be copied into this project directory, since updates of these files may affect the project. Often these files are already copied to the main project and these files can be linked from there. The files are:

- `GeneralRoutines.sun`
- `PlexTypes.sun`
- `CommandRoutines.sun`

These files are called packages, and holds procedures and type-definitions that makes the conversion easier. A copy of them or a link to them shall exist in the `lib` folder.

2. Fetch documents

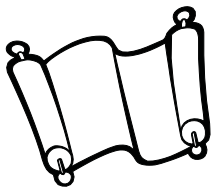
Get all documents related to the block that will be converted. The documents are

- `[blockname].ssurv`
- `[blockname].param`
- `[blockname].program`
- `[blockname]_ETI.script` (may be named different)

and should be stored in the `*_CAA107/src` folder.

3. Check validity

To avoid that time are spend on a block that is impossible to convert, an analysis of the block, by the designer, should be done to evaluate the block's complexity. Since inconvertible parts are hard to find, very good knowledge of both SDL10 and PLEX-C, and knowledge of the differences between the two languages is necessary. Help from block responsible and well experienced PLEX-C designers could be valuable in this step. Inconvertible parts may be implemented as inline expressions (PLEX-C statements inside the SDL10 code), but this is not desirable since this code cannot be tested in SDLtool. A redesign is then preferable, but takes longer time. If an inconvertible block is converted, the designer will be notified of the inconvertible parts during the process.



4. Analyze source code

The `[blockname].anapgm` file is needed by the reverse tool and is retrieved by analyzing the fetched files except the ETI (described above), in the order stated above. Errors that have risen during analyze must be fixed before continuing. Warnings are not necessary to correct, but some of them may need to be corrected in the converted code (at SDL level).

5. Test and adapt ETI

To test the converted block, when the conversion is finished (basic test), a correct ETI script is needed. Test the ETI and if needed, correct either the ETI or the source code (depending on fault) so that all test cases pass their criteria. Probably the ETI is correct, but one problem forces the designer to adapt the ETI. The problem denotes failure with the emulator and forlopp signals. This is solved by importing a block (MFM) to the dump and make changes in the ETI. The changes concern forlopp signals. The document [I7] describes this problem in more detail.

6. Copy files

To test the system after conversion, the files in step four and five are needed. Copy the following files from the `*_CAA107/src` directory to the `target` directory:

- `[blockname].ssurv`
- `[blockname].param`
- `[blockname].anapar`
- [blockname]_ETI.script (may be named different)

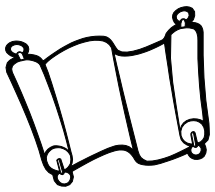
7. Arrange SDLtool Organizer

The environment in SDLtool's organizer must also be configured. The steps to do this are:

- Start with an empty document.
- Set the directories and mark the button for *Relative file names*.
  `Source Directory` is the same as where the system will be saved, i.e. the `sdl/src` directory.
  `Target directory` is were PLEX-C generated code should be stored, e.i. the directory named `target`.
- Add two new chapters, *Source Files* and *Generated PLEX-C*.
- Import `[blockname].program` and `[blockname].ssurv` files from the `*_CAA107/src` directory under the *Source files* chapter.
- Save the system with an appropriate name, such as `[blockname].sdt`.

Hints and comments

To understand the source code easier, the original *Flow Chart* is useful.



C.2.2 Convert signals

Purpose

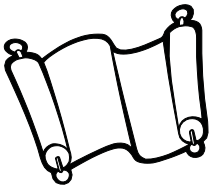
The signals used by the block must be converted into SDL signals. Much effort from the designer is needed and a fault introduced here will strongly affect the implementation at SDL level. For that reason the effort in this phase will be reflected in the total process time. The most important part is the `signals.data` file which holds information about all signals in the main project. It says what parameters in the PLEX-C signal that are *sending individual*, *sending blockreference* and/or *receiving individual*. More about these concepts in section Activities.

Input

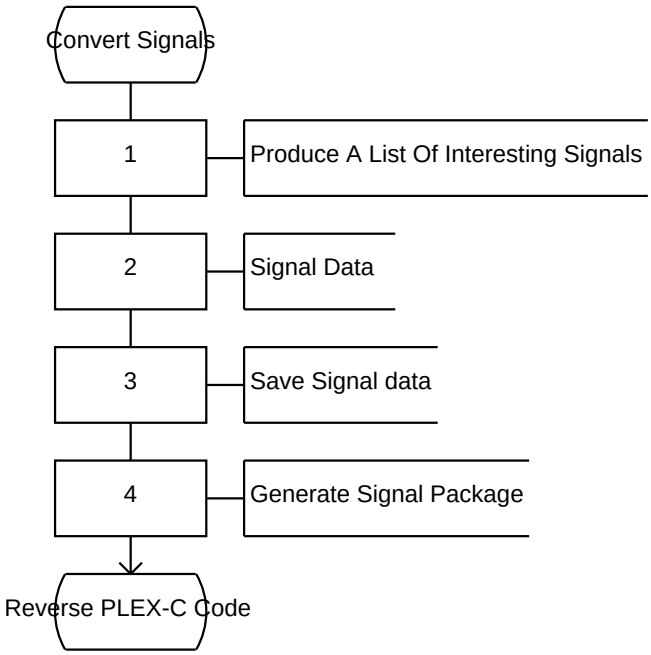
- `[blockname].program`
- `[blockname].ssurv`
- signals.data
- signal descriptions

Output

- signal package
- data in `signals.data`



Activities



1. Produce a list of interesting signals

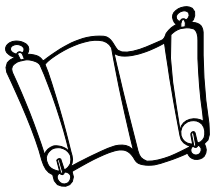
All signals in the signal survey are not of interest at SDL level and do not need to be converted. The reason is that some parts in the source code are automatically generated and do not exist in SDL. These parts are *Start/Restart*, *Size Alteration* and *Scanning individuals*. The affected signals are:

- `STTOR / STTORRY`
- `SETFS / SETFSEND`
- `GIVEFS / GIVEFSEND`
- `CONTFS / CONTFSEND`
- `CONTINUEB / CONTINUEC`
- `CLSCAN{n} / CLTIME0`

Remove these signals and all dummy signals (document number is dummy) from the signal survey imported in the Organizer. Make a copy of that signal survey, name it `signals.data`, remove all signals' document numbers and store the new file in the `sdl/src` directory. This file should include the remaining signal names on one line each and use this file when performing the following step.

2. Signal Data

For each signal, find out what parameters in the signal that is *Sending Individual* (SI), *Sending Blockreference* (SB) and *Receiving Individual* (RI). When doing this part, the focus should be on the block that will be converted. A parameter can only be one of the



types RI, SI or SB. RI and SI parameters are often called `TASKP`, which often the first parameter in the signal description. The parameter is RI if the signal is received by the block and SI if the block sends the signal. If the signal is both received and sent by the block, the parameter should be RI. Signals that have an RI parameter will be sent to the individual process in SDL, and signals without RI will be sent to the block process. Signal descriptions and comments in the source code often tells if a parameter is SB.

Any exact rules for what parameters that are SI, RI and SB does not exist, but the designer must know the block well and if necessary make some “trail and error”.

The parameter information is stored as numbers after the signal’s name in the `signals.data` file. The syntax is

```
<signal name> r b s
```

where **`signal name`** is the name of the signal, **`r`** is the parameter number that is RI, **`b`** is the parameter number that is SB and **`s`** is the parameter number that is SI. If a type does not exist in the signal, the corresponding **`r`**, **`b`** or **`s`** is set to 0. An example:

```
RMGETTARGDAT 1 0 0
RMGETTARGDATACK 0 0 1
RMGETTRANSMRES 0 2 1
```

This means that the first parameter in the `RMGETTARGDAT` signal is RI and SI in the other two signals. Parameter 2 is SB in the `RMGETTRANSMRES` signal. The zeros mean that there is no parameter of that type.

Do this step for all signals left in the signal survey.

3. Save signal data in `signals.data`

It is not necessary to update the global `signals.data` file that stores data about all signals ever converted. To avoid that unnecessary effort is put on signal conversions, i.e. retrieving signal data for one signal several times, an update is recommended. The permission rights to the global `signals.data` file is decided by the main project.

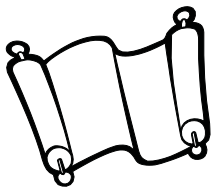
4. Generate signal package

- Mark the imported and modified signal survey in the Organizer.
- Select menu `Reverse -> Generate SignalPackage`.

If no errors were found, a signal package named `[blockname]signals` occurs in the Organizer, but if the generation was unsuccessful some files have to be removed before a new package can be generated. The affected files have the endings `.cif` and `.sun`.

Unfortunately the place where the corresponding package file is stored in the directory structure is incorrect. Do the following steps to correct that:

- Remove the package form the Organizer



- Move the file `[blockname]signals.sun` from the `src` directory to the `lib` directory (both under the `sdl` directory) since all packages (`.sun` files) shall be stored in here.
- Import the moved file under chapter *SDL Block Design.*

The package will appear again but it is now stored in the correct directory.

Hints and comments

When performing step two, some signals may already be examined and data about them are already stored in the `signals.data` file. Parameter data for these signals are not necessary to purchase, but since the data have been retrieved from another point of view, the data can be incorrect and has to be changed. It is recommended to examine all signals left in the signal survey and compare double signals.

C.2.3 Reverse PLEX-C code

Purpose

In this phase the reverse tool (PLEX2SDL) will be used to partly convert source PLEX-C code to SDL10 code.

Input

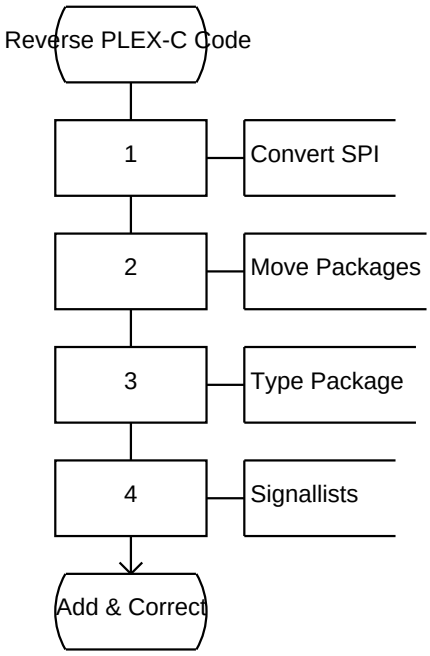
The following files:

- `[blockname].program`
- `[blockname].anapgm`
- `signals.data`

Output

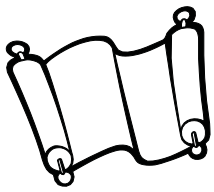
- An SDL block that partly describes the origin PLEX-C block.
- file `[blockname].cov`

Activities



1. Convert SPI

- Mark the `[blockname].program` file in the Organizer.
- Select menu `Reverse -> PLEX-C to SDL`.



The system will be busy for a while and meantime its status will be presented in the log window. After a couple of minutes (depending on the block's size) the automatic conversion process is done and errors and warnings may be presented in the log window. If errors occur they must be corrected before the block can be correctly converted. No SDL code is generated. The designer should be aware of the warnings and if possible correct them, but it is not necessary before continuing the process. If this step is re-executed, some files have to be removed first. Type the following command in the `sdl/src` directory:

```
rm -i *.ssy *.sbk *.spr *.spd
```

In the file `[blockname].cov` unconverted statements and statistics are presented. This file is used to manually convert the untreated statements. These statements are called *white spots*.

2. Move packages

In the Organizer lots of symbols will appear. The new packages are incorrectly stored in the directory structure and have to be moved. Remove the packages `plextypes` and `GeneralRoutines` from the Organizer and delete corresponding files in the `sdl/src` folder (the files are linked). Import the removed packages from the `lib` folder in the Organizer under the same chapter as they were before they were removed.

3. Type package

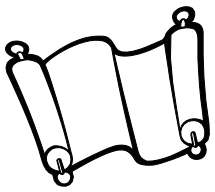
In the block, structured variables will receive values from signal parameters, and due to that the signal parameters must also have the same structure. That is the reason for moving out type definitions to a separate package. It is suggested to use a "standard type definition", an already defined type package that is standardized to avoid type conflicts between blocks, especially the signal parameters must have some "standard" structure which is defined by the signal coordinator.

The signal package must also be referenced by the block.

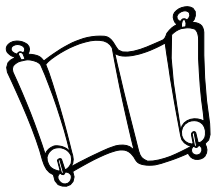
- Add a new package to the system and call it `[blockname]types`.
- In the type definitions for the block, cut all except the synonym definitions.
- Add the cut definitions in a new text box inside the previously added package.
- Add a reference to the `plextypes` package in the new package by typing `USE plextypes;` in the *package reference* symbol.
- Add a reference to the new package in both the signal package and in the system level layout, by typing `USE [blockname]types;`.
- Add a reference to the `[blockname]signals` package in the system level layout.

4. Signallists

Hopefully the reverse tool will construct all signallists, but sometimes it won't (an error reported to Telelogic). Check if all signallists (four) exists in the system level. If not, they have to be made manually by checking where each signal is going, i.e. to/from individual



or block process. Both the `signals.data` file and the *signal survey* are helpful here. Signals with `0 0 0` in the `signals.data` file goes to/from the blockprocess, and most of the others goes to/from the individual process. The signal survey is used to see the direction of the signals.



C.2.4 Add and correct

Purpose

The parts that the reverse tool did not convert have to be converted manually. Most of these whitespots are easy to convert, and some of the untreated statements are automatic generated by the PLEX-C generator (`SDL2PLEX`).

There will also be type conflicts that has to be resolved.

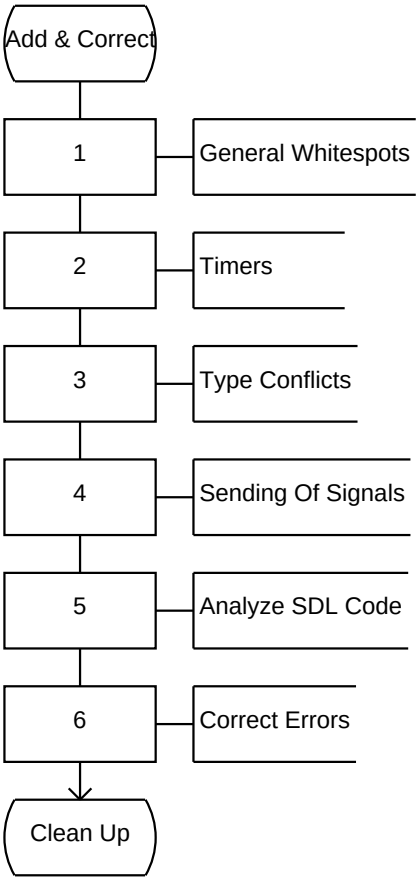
Input

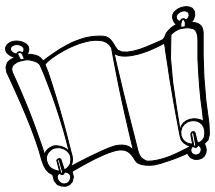
- Partly correct converted SDL block.

Output

- More correct SDL block

Activities





1. General Whitespots

The code that were not converted into SDL are called whitespots. These whitespots are mostly automatic generated code which needs directives to be correct. The reverse tool's work instruction [I8] page 62 and forward, describes how to recover whitespots. The steps in the work instruction are related to most of the untreated statements described in the `.cov` file. Some whitespots may be:

- **Size Alteration** - except for the steps in the reverse manual, check if there is a seize[1] of an individual or the individuals are mapped between blocks. If the latter, the individual processes are not created dynamically, but are started directly and for that reason the number of processes at start, and the maximum of processes are set to the value that the `MAX` directive has.
- **Process start / creation** - as described for size alteration, it is not always necessary to create individual processes. If the individuals not are created, they must never be terminated, instead they should jump to the `IDLE` state.
- **Timers** - for more information in addition to the reverse manual (see below).
- **Forlopp handling** - Often comments in the SDL code says what should be implemented.
- **Start / restart** - Global start / restart procedures are already constructed, but for the individuals they have to be made manually.

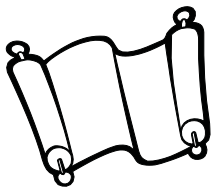
2. Timers

The automatic generated code for *Scanning Individuals* are only generated if *Timers* are used. For that reason timers on PLEX-C level must be converted to SDL timers. The timer variable at PLEX-C level is often called `TIMER` (or similar). All statements (in SDL) where this variable is set to a value (use search in SDL editor) have to be replaced with the SDL timer instruction `SET` or `RESET`. Beware of that type conflicts may occur, but this problem is solved by using the subprocedure `ConvertToDuration`.

When a timer expires, a signal is sent to the process. The signal is called the same name as the timer. In PLEX-C timeout signals are often called `ITIMEOUT{n}`. Either the timers must be named after these signals, or the receptions of the timeout signals must be changed to the timers name. Which one of these the designer choose does not affect the system at all.

The timer functionality is implemented in PLEX as a variable which increases periodically. The value of the variable may never be zero (it is used in divisions), and for that reason its initial value is one and it is periodically increased with the value two. Remember that PLEX-C variables are positive and cyclic, i.e. $\texttt{FFFF}_{hex} + 1_{hex} = \texttt{0000}_{hex}$. The timer functionality which the SDL2PLEX generator implements works different. Instead of increasing with the value two, the value one is used. To avoid division with zero, a condition is setup before each increase, and if the value is $\texttt{FFFF}_{hex}$ it is changed to one, other-

1. Seize an individual means allocate resources for it.



wise an increase is performed. This means that it will take two times longer until the timers expires. This problem is solved by changing the assignments of the timer length variables to their half values. These variables are assigned in the beginning of the program (start transition of block process) and are used when setting timers in the code.

3. Sending of signals

This step concerns only sending of signals with a specified receiver, i.e. a send signal symbol with the *to* keyword. The reverse tool has difficulties to convert the receiver correct, wrong or unassigned variables are used. This fault is not found by neither the SDL analyzator nor the PLEX-C generator, but maybe in Basic test, i.e. the fault may not even be found within this process.

- Look up all sending of signals with the *to* keyword and check if the receiver is correct (correct and assigned variable).

4. Type conflicts

Between signal parameters and variables, type conflicts will occur, because all signal parameters are declared as `Nat16`. This is easy to solve with help of the analyzer. This is explained further in step 6.

5. Analyze SDL code

- Mark the system and select `Generate -> Analyze`.
- Select *Syntactic analysis*, *Semantic analysis* and *Check output semantics*.
- Press *Full Analyze.*

6. Correct Errors / Warnings

Errors highlighted when analyzing the system must be corrected before continuing. Take notice about the warnings, and if possible correct them too.

When converting signal parameter types, the structure of a parameter should be similar to the structure defined by the signal description. Variables in the block that have some connection to a signal parameter should have that signal parameter's structure, and if that is impossible, macros for converting between structures must be implemented.

When some (or all) errors / warnings are corrected, go back to step 5 to analyze the system.

Hints and comments

After first analyze, use the Analyze quick button.

C.2.5 Clean Up

Purpose

This phase intense is to clean up unused variables and make the code more readable.

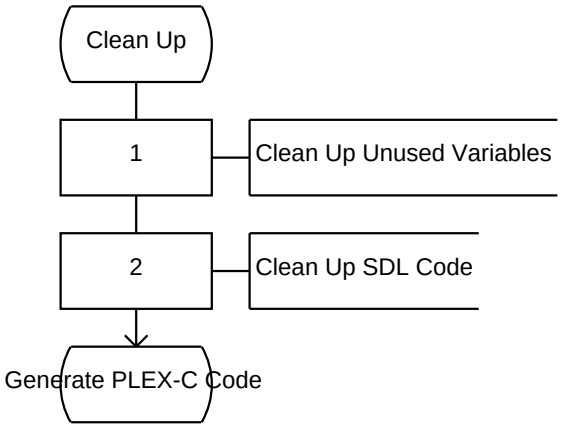
Input

- Fully analyzed and corrected SDL block

Output

- Cleaned up SDL block

Activities



1. Clean up unused variables

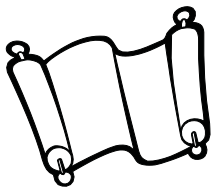
- Mark the system and select `Generate -> Analyze`.
- Select same options as in phase 4 step 5, but also select *Check unused definitions*.
- Press *Full Analyze.*

Warnings reported on the `plextypes` and `GeneralRoutines` packages, and on unused start/restart procedures shall be ignored, but all other warnings should be fixed. Since all errors were fixed in the previous phase, non should be reported here.

2. Clean up SDL code

This step is optional, but should be used to make the code easier to follow and understand.

- Go through all pages in the SDL code and make layout improvements described in [I9].



C.2.6 Generate analyzed PLEX-C code

Purpose

In this phase analyzed PLEX-C code will be produced, and also an adjusted and analyzed signal survey. Note that *all referred files are stored in the target directory.*

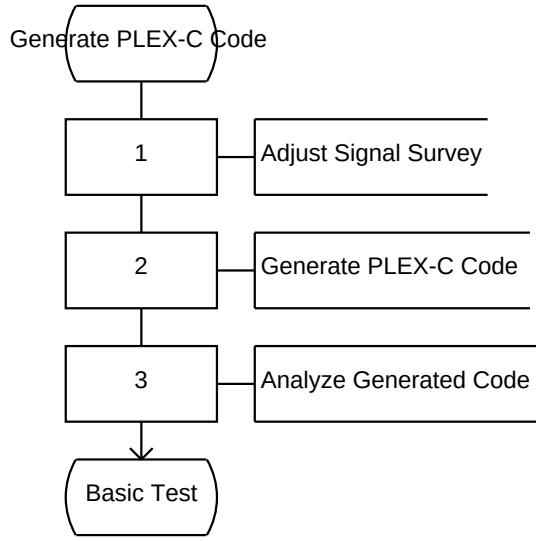
Input

- analyzed SDL block
- original signal survey
- original parameter file (`.param`)
- analyzed parameter file (`.anapar`)

Output

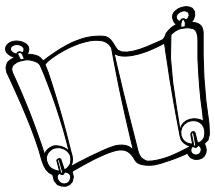
- Analyzed PLEX-C code
- adjusted signal survey
- analyzed signal survey

Activities



1. Adjust signal survey

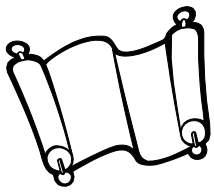
- Replace signals `CONTINUEC` and `CONTINUEB` in the signal survey with:
  `CONTINUECSAE  71168 / 155 14 - ANT 292 01`
- Analyze the signal survey



2. **Generate PLEX-C code**
   - In SDLtool: Mark the system and select `Generate -> Make`. The Make window appears.
   - Make sure that *Analyze and generate code* is selected, and *Makefile* and *Compile & Link* is deselected.
   - Select `PLEX` as *Code generator* and make sure that the *target directory* is correct according to phase 1. Click on *Full Make*.
   - If errors occur, correct them and restart from phase 4 step 5, otherwise continue.
   - Import the generated code under the *Generated Code* chapter.
   - Remove SDT references.
   - Break lines longer than 112 characters in the generated code (otherwise the analyzer will dead lock).
3. **Analyze generated code**
   - Analyze the generated PLEX-C code (willingly with the compile option)
   - Found errors shall be corrected in the SDL code, and then restart from phase 4 step 5.

Hints and comments

Lines longer than 112 characters are often `TRANSFORM` statements.



C.2.7 Basic Test

Purpose

Adapt the ETI to the generated PLEX-C code, and check that the conversion process not implemented any faults.

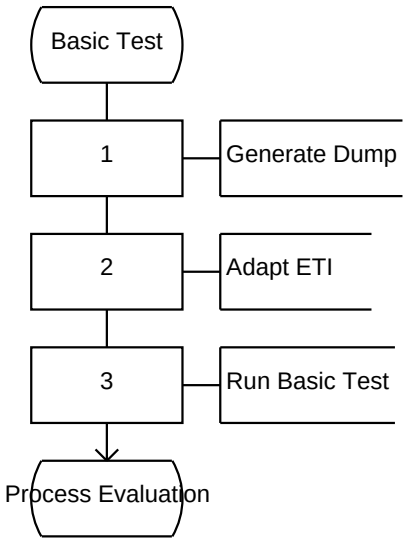
Input

- ETI script that is OK towards the origin source code

Output

- SDL block fully PLEX-C basic tested
- Adapted ETI

Activities



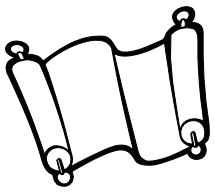
1. Generate dump

Generate a dump of the source code as in usual PLEX-C development. Remember that if the MFM block was needed before, it is still needed (forlopp problems with emulator).

2. Adapt ETI

Variables have changed names with the conversion. This must be changed in the ETI. Some of the affected variables are

- `CINDUM-> CXGEN[process name]NUM`
- `COWNREF-> CXGENOWNREF`
- `STATE-> XGEN[process name]STATE`



- `OCCUPATIONFLAG-> XGEN[process name]STATENOTIDLE`
- `CCLOCK-> CXGENCLOCK`
- `CSCANNING-> CXGENCLOCKPERIOD`
- `CPREPNUM-> CXGENMSMTAPDATAPREPNUM`
- `FLCONNFIDTASK-> XGENTASKDATAFLCONNFID`
- `CFLSTATUS-> XGENTASKDATAFLSTATUS`

Note that the old variable name may differ. Rest of the affected variable names have to be found out manually by “trail and error”.

If the ETI enumerates the states, the order of them is important. Check in the generated PLEX-C code where the states are enumerated, the same order must be used in the ETI.

When checking timers, their names are the same as in the SDL environment.

More adaptation hints are described in [I8], page 142 and forward.

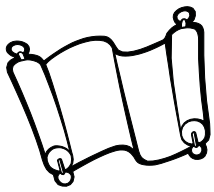
3. Run Basic Test

Run basic test as in usual PLEX-C development. Errors found may have its origin either in the SDL code or in the ETI. If the latter, correct the ETI and restart this step, otherwise correct the fault at SDL level and restart from phase 4 step 5.

Note that some errors can not be corrected though. It is errors according to the automatic generated code. In this case *Trouble Reports* must be written and delivered to Telelogic.

4. Hints and comments

To find errors that was found in basic test, one can simulate the block in the SDL environment. If you choose to simulate in the SDL environment, and if individuals are created during start up, don’t forget to temporary change the number of individuals to a few, otherwise the system will be very slow.



C.2.8 Process evaluation

Purpose

Evaluate and improve this process.

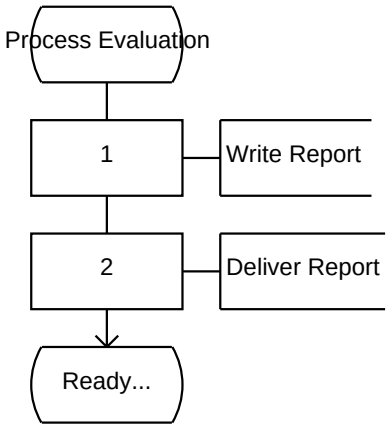
Input

- Experience of this process.

Output

- Improved process.

Activities



1. Write report

Write a report on basis of collected comments and experience from an execution of this process.

2. Deliver report

Send the report to the process owner so he/she can update the process.

Hints and comments

Make notes while running the script.