Restructuring SDL to Improve Readability

Ole Henrik Dahle

May 6th, 1999

Institutt For Informatikk, Universitetet i Oslo

Preface

This thesis is part of a Cand. Scient. (Master of Science) degree at the Institute For Informatics, University of Oslo. The theoretical studies were done in 1998, and the programming and writing was done during the fall and winter of 1998/99. The programming and experiments were done at Ericsson NorARC (Norway Applied Research Center).

I would especially like to thank my supervisor, Øystein Haugen, for being a great help in my work. Not only has he guided and encouraged me, but also helped me with many practical things. I would like to thank all the people at Ericsson NorARC for letting me work there. I would also like to thank Hans Steller and the people at Ericsson's PSTN (Public Switched Telephone Network) group for letting me participate in their TTM3 project.

Contents

Abstract

This thesis shows how to use techniques and theory from program transformation to improve readability of SDL (Specification and Description Language) diagrams. An introduction to software reengineering in general and restructuring and transformation in particular is provided to establish the relationship between transformation and pretty printing. I then introduce a set of rules for readable SDL. To demonstrate the applicability of these rules, I have created a prototype program that enforces the rules. A real world example, from a project in Ericsson, is fed through the program and the results are evaluated.

Keywords: Restructuring, program transformation, pretty printing, SDL, readability

Executive summary

Program transformation is the process of rewriting one program into another using a set of transformational rules. A transformational rule has a left-side pattern and a right-side pattern, and possibly some conditions. When the left-side pattern matches a string in the source program and the conditions are satisfied, the string is substituted with the instantiated right-side pattern.

Restructuring is transformation of a description to make it easier to understand or less susceptible of errors when future changes are made [Arnold89].

In this thesis, the theories of program transformation and restructuring are used to improve readability of SDL (Specification and Description Language) diagrams. The rules used are taken from Bræk and Haugen's "Engineering real time systems" [Bræk93], with some additions by me. The majority of the rules concern only diagram layout issues, and do not alter the semantics of the SDL. The notable exception is the rule that discourages use of connections, the "GOTO of SDL". If the user allows it, connections are expanded or turned into procedures.

The major rules I selected are as follows:

- Only one state per page
- Source / destination should be specified on all signals
- Avoid connections, use procedures instead
- Branch on signals, not decisions
- Use meaningful names, not shorter than five letters

I have made a prototype program in Perl that implements these rules. The program works on SDL descriptions in PR/CIF (Z.106) format, which can be read and written by most SDL tools. The program worked satisfactory with Telelogic's SDT, Cinderella, and Verilog's Geode.

The program was tested on a real world example, an SDL process from a project in Ericsson.The results were positive, and confirmed that program transformation is a viable option for improving readability.

About the thesis

Motivation

When I started on my Cand. Scient. degree, I planned to work on code reuse. My preliminary question was “how and why does old code hinder adoption of new techniques?” To study this, I worked at NorARC and learned their development methodology. I also participated in a redevelopment project using SDL for design. However, this work did not give me enough background data to say anything meaningful about old code and adoption of new techniques.

On the other hand, through my work I got involved with writing Perl scripts to reformat text files. During my study of the reengineering literature, I got interested in program transformation, and thought Perl could be a good tool for this. When I saw how SDL was written as text files in the PR/CIF format, I realized that in this format, SDL could be transformed by a perl script without much trouble.

The step of preparing reverse engineered or automatically generated SDL for forward engineering had not been looked much into, in Ericsson or in the literature. This provided me with an opportunity to use my perl skills to try out transformational techniques to improve readability, something that had not been done on SDL before.

Objective

The objective of this thesis is to show that it is feasible to automatically restructure SDL to achieve better readability, and to support this activity with theory from program transformation.

I will try to meet this objective in four phases:

1. I will present the basics of reengineering and program transformation, and link it to automatic restructuring.
2. I will present a collection of rules for readable SDL that can be applied automatically

3 I will construct a program that uses these rules on SDL diagrams

4 I will test my program on a real world example

Organization

Chapter 1 is an account of my own experience with software maintenance. The chapter is meant to introduce the reader to the problems of maintenance.

Chapter 2 contains a introduction to the field of reengineering. The goal of this chapter is to familiarize the reader with the jargon of reengineering and show how reengineering can address the problems presented in the first chapter.

Chapter 3 presents the basic facts of program transformation, and together with chapter 2 fills out the theoretical background for the thesis.

Chapter 4 presents the rules for readable SDL and how restructuring of SDL can fit into different development models.

Chapter 5 describes the implementation of the prototype program. This is done so that the reader can see how the transformational theory is applied, and how the rules can be implemented.

Chapter 6 describes the experiences I made when I used the program on a real world example.

Chapter 7 sums up the results, and in Chapter 8 I present my own conclusion.

Appendix A contains a dictionary for the thesis. It includes the definitions of the terms used in the thesis. The definitions are not repeated in full in the text, so the reader is encouraged to look up terms in the dictionary.

CHAPTER 1

EXPERIENCE FROM SOFTWARE MAINTENANCE

This thesis deals with reengineering and restructuring as an aid to software maintenance. To help the reader understand some of the problems associated with maintenance, this chapter describes a maintenance job I did and the problems I encountered.

The task

In October 1998, I was given the task of altering a Perl script to do some new things. Although the job was rather small, (the script was only about 200 lines, and the job took about 20 work hours) I think it can illustrate some of the common problems in maintenance.

The script was written by Nils Faltin at the university of Erlangen, Germany. I had no contact with him, and there was no written documentation, so I had only the source code (with some comments) to go by.

The purpose of the script was to extract BNF rules[1] from an ASCII document and make a cross-referenced HTML index of the rules. The original version was divided in two parts. The first script, `extract.perl`, would

1. The Backus-Naur Form (See definition in dictionary, appendix A).

scan through a text file and write all the rules found to a file called `rules.bnf`. Then the second script, `indexbnf.perl`, would read the rules from the file, build a table of which non-terminals the rules used (from here on called the used-by table), and write a hyperlinked list of all the rules to a HTML file. The reason for this division was to let the user correct the rules in `rules.bnf` before the list was generated.

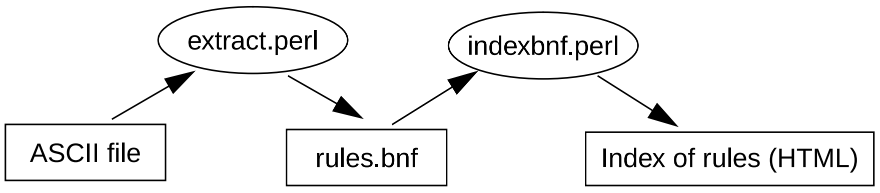


Figure 1.1 Source and products of the original scripts

The customer for this job thought the original script was good, but wanted to have the hyperlinks put into the document itself, instead of in a separate file. The source document was to be in HTML, not ASCII. The customer also wanted the two scripts joined into one, because he thought it would be more user friendly with only one program. If the first part misinterpreted the rules, he would rather alter the source document than to rework the intermediate results in `rules.bnf`. The script was called `linker.pl`, because its primary function was to create links inside the HTML document.

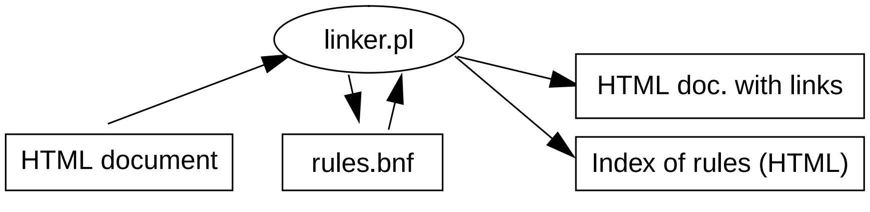


Figure 1.2 Source and products of the new script

This made the task a case of both adaptive and perfective maintenance. (See dictionary for definitions). On one hand, the mission was to adjust the script to a new environment (HTML instead of ASCII). On the other hand, the script would be extended with a new feature, inserting links into the source document.

The document in focus was an ITU (International Telecommunications Union) specification of the MSC (Message Sequence Chart) language, with a couple of pages of text at the start, and then the specification of MSC in BNF rules.

Being a little bit naive, I thought I could make the script fullfill the new requirements with just minor modifications. I did not really understand the part of the script that read the rules from `rules.bnf` and created the used-by table, but instead of using a lot of time to understand it, I decided to use it as a black box.

The work with the script went through three major iterations and approximately ten minor versions, and was shown to the customer five or six times. The reason for the many deliverances was that altering the script was done quite quickly, so a prototyping approach was possible. The requirements for the script changed a little from version to version, as the customer and I discovered what was possible and favourable. This made the script go through a bit of evolution, and not surprisingly, the evolution followed Lehman's law of increasing complexity (see Lehman's laws on page 13).

The combined length of the original two scripts was 208 lines, including the printing of a large header. The first version of my script was 177 lines long, the final 299 lines. The cyclomatic complexity[2] of the original script was 22. The first version of the new script had a cyclomatic complexity

2.Cyclomatic complexity is a measure of the number of independent paths through a program, devised by McCabe [McCabe76]. If a program is drawn as a graph with the statements as nodes and decisions as the edges between them, the cyclomatic complexity is: CC = Number(edges) - Number(nodes) +1. If there are no GOTOs in the program, the cyclomatic complexity is simply the number of conditions +1.

of 20, but in the final version it had grown to 38. The increase was mainly caused by a growing number of nested if-statements, taking care of more and more special cases.

The first iteration

To begin with, the script handled all lines containing '::=' as BNF rules. This was not acceptable, as the '::=' expression was used in some textual areas of the document as well. To get around this, I added an extra test, checking if the line began with '`<P>&lt;`*`name-of-rule`*`&gt;`', since all the BNF rules were written this way.

The original script expected the non-terminals to be enclosed with less-than and greater-than signs ('<' and '>'). In HTML, all the tags are enclosed with these signs, and less-than and greater-than signs in the text are coded as '&lt;' and '&gt;'. To make the BNF rules appear correctly in `rules.bnf`, I stripped all the HTML tags from the rules found in the source document, and replaced '&lt;' and '&gt;' with '<' and '>' before the rules were written to the file. This added a couple of lines to the script to do the stripping. As a side effect all formatting of the text in the rules was lost.

This was irritating, as boldface and underlined characters were used in the rules to convey special meaning. For example, if the words before "name" or "identifier" was underlined, they should not be linked to. It looked like this:

```
<shared instance list> ::= <instance name> [ , <shared instance list> ]
```

The original script would create links to everything between '<' and '>', but a link to "instance name" would be wrong. I created a temporary solution to this by replacing the underline-tag with a pound sign during tag stripping, and then ignoring terms with pound signs in the linking phase.

The stripping of HTML tags also caused problems for graphical rules. These rules had to include a HTML tag to show the picture of the symbol. In the first version, the tag was considered a rule by the script, which produced a lot of nonsense. To fix this, I had to create a hack (a fix that does the right thing, but breaks some rules doing it, and often is difficult to maintain). If the script detected the string “IMG SRC” in the rule, it knew it was a graphical rule, and made a hyperlink only for the left term of the rule.

If, on some later occasion, a BNF-rule contained the words “IMG SRC”, the rule would be considered a graphical rule instead of a textual. This error would probably be difficult to figure out for another programmer doing maintenance on the script.

The second iteration

When the customer tested the first version, he was basically satisfied with the script, but pointed out the problem with ‘identifier’ and ‘name’ statements. After browsing the HTML document produced by the script, he also found it awkward to have the index of the rules in one document and the definitions in another.

The customer also wanted a BNF grammar describing the allowed format of the input to the script. This would make it easier to rework other documents to work with the script.

In the second iteration of the script I experimented with different ways to show the index of the rules and the definitions at the screen simultaneously. First I made the script write an index of all the rules at the end of the document. Then I tried having the definitions and index in two different windows, but neither solutions worked very well for the customer.

To try to remedy the ‘identifier/name’ problem, I had to move the underlined words outside the ‘<‘ and ‘>’ signs, and then let the linking go as usual. A line that originally looked like:

```
<shared instance list> ::= <instance name> [ , <shared instance list> ]
```

was changed to:

```
<shared instance list> ::= instance <name> [ , <shared instance list> ]
```

with the correct link to the definition of ‘name’. I thought it was a good solution, but apparently it broke the rules of BNF, and the customer did not like it.

This version also included a BNF grammar of the allowed input, but since it was my first attempt to describe anything in BNF, it had some shortcomings.

As the recognition rate on the rules approached 100 per cent, I noticed that there were actually more substitutions than rules. That would have to mean that some terms were defined more than once. Since the associative array that stored the rules was indexed by the name of the left terms, a second definition of a term would simply overwrite the first one.

After examining the document closely, I found that in the beginning of the text there were some examples with left terms identical to real rules, but with simplified right sides. When the script went through the document, the right sides of the examples were substituted with the right sides of the real rules, destroying the examples. This problem had to be solved to make the script usable.

The third iteration

The goal of this iteration was to stop the experimentation, weed out the bugs, and produce a version that recognized all the BNF-rules, and without ruining anything else in the document.

The prime concern in this respect was of course the problem with the examples. Multiple definitions of a term is against the Backus-Naur Form, but I could not force the author of the document to use other examples. I had to find a way of identifying the examples and ignore them. The solution I came up with was simple, but brutal: Until the script read the words “concrete textual grammar”, it would not try to recognize any BNF-rules. This did the trick, since this phrase was used only directly before the rules. It worked fine on the document at hand, but obviously this was a terrible hack, and there is no guarantee that it will work on any other document.

At the time of the third iteration, I had got quite fed up with the stripping of HTML tags and the side effects it created. I gave up on the black box programming and read the code that creates the used-by table thoroughly.

As it turned out, all I had to change was the regular expression that identified the terms in the rules. Instead of demanding that terms should be enclosed in less-than and greater-than signs, I made the expression accept terms enclosed by ‘&lt;’ and ‘&gt;’. In a flash, the need for stripping HTML tags was gone, and the script worked a whole lot better.

In my third attempt at making the results display nicely, I put the index and the definitions in two frames within the same window, in the common menu/content style. This was OK with the customer, and the display was not changed any more times.

The script was accompanied by the second version of the BNF grammar for allowed input. It looked like the following:

```
<document> ::= (<text> | <production>)*
<text> ::= <startline> <.>* <endline>
<production> ::= <leftside> <rightside>
<leftside> ::=  <startline> ‘<P>’ <term> < >+ ‘::=<br>’ <endline>
<rightside> ::= (<.>* | <term> | <endline> <startline>)* <endse-
quence>
<endsequence> ::= ‘</P>’ <endline>
<startline> ::= the start of a line
<endline> ::= the end of a line, the newline character \n
<term> ::= &lt; <a>* &gt;

<.> ::= any character
<a> ::= an alfanumeric character, or ‘-’
< > ::= a blank
‘ means a quote - the text inside ‘s has no special meaning
+ means zero or one time
* means zero or more times
```

As this described the input reasonably well, it was accepted by the customer.

After the second version of this iteration, the script was declared “good enough” by the customer, and the document produced by the script was distributed to the group of persons that also had received the source document. The response to the product was positive, but I’m afraid that the new version of the script is considerably less maintainable than the original.

Some typical maintenance problems

This relatively small maintenance task was definitely a learning experience for me. I made a lot of basic mistakes that I've only read about and never thought I would do in practice. The most basic mistake was to not assess how the new environment (HTML input instead of ASCII) could create problems for the script.

My problems seem to correspond, to some degree, to the common problems in maintenance. In maintenance environments, there is typically a large backlog of change and enhancement requests. There is a constant pressure to meet as many as possible of these request with each new release of the software.

Lack of planning

When there is little time available to deal with each request, it is only human to cut down on the efforts that have small short term consequences. Planning development and evaluating future effects of the change is easy to cut down on. This lack of planning also happened to me, but probably more out of laziness than hurry.

Limited understanding of the system

The time spent reading and understanding the code is also naturally to cut to a minimum. The maintainer will, if in a hurry, study the code until he/she knows how to implement the change, no longer. In my little project, this had grave consequences, as most problems arised from the part of the code I did not read carefully enough.

My attempt at black box-programming undoubtedly made the script more complex and buggy. During the three iterations, the script grew in size from 177 to 299 lines, despite that the functionality did not change much.

Unstructured execution

Generally, the process would have improved a lot with more structured execution. Because the task always seemed “almost finished”, I adopted more and more of a “code and fix” development model. Especially in the end, when I was running out of time, I did not care too much if my code would be reusable or maintainable. I just wanted the script to work, no matter which hacks I had to create.

When the development time is short, full-fledged development models with multiple stages seem like too much overhead for the maintainer. A solution is found quicker with the “code and fix” approach. All this leads to the complex, bug-ridden, poorly documented systems predicted by Lehman’s laws.

My problems, and perhaps most maintenance problems, seems to stem from extending and altering programs little by little, without seeing the greater picture. If the maintainer took a step back, and analysed the problem from the beginning, perhaps the solutions could become more consistent and robust. An activity that tries to do this is reengineering.

CHAPTER 2

AN INTRODUCTION TO REENGINEERING

Background

As described in the previous chapter, hastily performed maintenance usually degrades the quality of software. If anything but trivial change is required, the design of the system should be rethought, and the system rebuilt to fit the new requirements. But to throw away the systems and develop new ones may not be a viable option. This will invariably cost a lot of money for the organization, and the time required to develop new systems from scratch may be too long.

A possible way of accomplishing dramatic change for less-than-dramatic expenses is software reengineering. The essential difference between normal maintenance and reengineering is that reengineering takes a step back and analyses the system over again. Maintenance is concerned with making a fix work, whereas reengineering is concerned with what the system **really** should do, and how.

Reengineering is the process of understanding and altering existing computer systems. (See Appendix A for a more precise definition.) Reengineering usually involves some kind of reverse engineering, where information of higher levels of abstraction is extracted from the implementation of the system at hand. This information is then the basis for the further steps of the reengineering process. The information can be restructured to clean up complex and messy systems, or can be altered to incorporate new functionality. Then a new implementation is made from the information. This forward engineering step is done as in normal software development.

With the increasing number of legacy systems around, reengineering is becoming an option to consider for more and more organizations.

According to Robert S. Arnold, there are at least seven reasons why reengineering is important [Arnold93]:

1. Reengineering can help reduce an organization’s evolution risk
2. Reengineering can help an organization recoup its investment in software
3. Reengineering can make software easier to change
4. Reengineering is big business
5. Reengineering capability extends CASE tools
6. Reengineering is a catalyst for automating software maintenance
7. Reengineering is a catalyst for applying Artificial Intelligence (AI) techniques to solve software engineering problems

In this paper, we will look at reengineering from a maintenance point of view, and from this perspective point number one, three and six will be most interesting for us.

Point one: When a organization’s reengineering practice matures, reengineering will be a less risky path for evolution than new development or perhaps even normal maintenance. New development is often expensive and uncertain. Normal maintenance, unless carefully carried out, tend to make the software more complex and less reliable.

Point three: Reengineering can help the maintenance programmer understand the software more quickly. Also, the alterations can be done at design level instead of in the source code.

Point six: Reengineering tools can become largely automatic in the future. With good tools for program analysis, design recovery, and code generation, the maintenance programmer can work on informative abstractions of the system. This brings the vision of automatic programming to maintenance programming: The developer should be able to decide what the program should do, and then the implementation will be generated automatically.

History

The art of reengineering has existed as long as the art of software development. There has always been a need for reversing the steps of development and do things over again, but this has not been considered a field of its own. In fact, the word “reengineering” was not widely used before the end of the 1980s.

In 1980, Manny Lehman published his five laws of software evolution [Lehman80]:

1. Continuing change
   A program undergoes continual change or becomes progressively less useful.
2. Increasing complexity
   As an evolving program is continually changed, its structure deteriorates.
3. Fundamental law of program evolution
   Program evolution is self-regulating, with statistically determinable trends and invariances.
4. Conservation of organizational stability (invariant work rate)
   During the active life of a program, the global activity rate in a programming project is statistically invariant.
5. Conservation of familiarity (perceived complexity)
   The content of an evolving program’s successive releases is statistically invariant.

These laws are almost considered “the natural laws” of software maintenance, and are the fundamentals of the vocabulary used in software maintenance research.

The first and second laws are especially interesting for software reengineering. The first law says that systems will never be complete, that they will always need to change and evolve as long as they are used. Therefore systems should be made with evolution in mind. It is foolish to expect that a program will not need changes, no matter how perfect it fit the specification on the day it was released.

The second law tells us that, as they evolve, large systems grow more and more complex. The complexity may increase because problems are corrected in an ad-hoc manner, without trying to make the changes consistent with the rest of the system. Another reason may be that the maintainers do not fully understand the program they are changing (because of poor documentation in the code and on paper). Therefore they add code to work around problems in the original code instead of attacking the problem’s root. The challenge for reengineering is to break this law, at least for short periods of time, so that the complexity in old systems can be reduced, and the life of the system lengthened. Shari Lawrence Pfleeger discusses the rest of Lehman’s laws in her article “The Nature of System Change” [Pfeelger98].

Since reengineering is often used in software maintenance, reengineering research was long considered a part of the maintenance research field. Maintenance journals and conferences have often presented good works on reengineering. The “Conference on Software maintenance”, which has been held since 1984, has had sessions on reengineering for reuse, restructuring and such. The “Journal of Software Maintenance: Research and Practice”, which has been published since 1989, also has many good articles on reengineering.

In the second half of the eighties, there was a growing recognition of reengineering as a research field in its own right. The word “reengineering” began to be widely used, probably influenced by the buzz around Business Process Reengineering. Personally, I consider January 1990 to be the birth date of software reengineering as a area of its own. At that time the journal “IEEE Software” published a special issue concerning reengineering. The issue contained 6 articles about reengineering, some of them among the most referenced articles in the field.

Especially important was Chikofsky and Cross’ article “Reverse Engineering and Design Recovery: A Taxonomy” [Chikofsky90]. The article proposed definitions on most of the terminology of reengineering, and laid the groundwork for the literature about the subject. Although it has not been officially endorsed as a glossary, it is commonly accepted as the authoritative definitions of terms, and most works follows its guidelines for the use of the terms.

During the nineties, the community has consolidated. A lot of articles and books have been published, and different forums has been established. In 1993, Robert S. Arnold published the book “Software Reengineering” [Arnold93]. The book contains a collection of important articles from the different areas of research, with introduction to reengineering and the specific areas by Arnold. The most important meeting places for the reengineering community is the “Reengineering Forum” conference, which has been held since 1993, and the “Conference on Reverse Engineering”, which was first held in 1994.

Areas of research

The topics of the reengineering research have always mirrored the topics of the development community. A typical topic to discuss in articles and at conferences is how to modernize programs to fit the programming paradigm of the time.

In the 60s and 70s, transforming unstructured programs into structured ones was an important issue. Already in 1966, Böhm and Jaccopini wrote an article on structuring program flow diagrams [Böhm66].

From the beginning of the 1980s, the interest in reengineering COBOL programs arised. Large numbers of sizeable COBOL systems had been made in the late 60s and 70s, and was not according to the structured programming paradigm. This is still a hot topic, and it seems like the reengineering of COBOL will peak in 1999, with the preparation of these systems for the millennium change.

From the mid-eighties, there has been an interest in migrating applications from mainframes to client/server architectures. The trend was started by the technological development; workstations and PCs took over for dumb terminals. With so much processing power at the desktops, it was no longer cost-effective for applications to run only on the servers. Ganti and Brayman [Ganti95] emphasize that business processes should also be reconsidered when migrating to client/server solutions. This shows that software reengineering can be an integral part of Business Process Reengineering (BPR).

From the late eighties and into the 1990s, a popular topic of discussion and research has been the reengineering of functional oriented programs into object oriented ones. Ivar Jacobson gives a method for reengineering parts of a system into object oriented modules in his article from OOPSLA 91 [Jacobson91]. His solution is to first create an object oriented domain model by design recovery and domain analysis. Then he reimplements selected parts of the system with OO technology. In addition, he creates OO wrappers so that the interfaces between the old and new parts are completely object oriented. By using this method, Jacobson argues, it will be possible to gradually modernize a legacy system to a object oriented one.

One of the newest trends in reengineering is patterns. Because of the great interest the software development community has shown to patterns, researches have wondered what patterns could contribute to reengineering. Stevens and Pooley [Stevens98] argues that reengineering patterns could help (re)developers choose a line of action for their projects, but that patterns cannot replace a methodology. In this sense, reengineering patterns differ from design patterns in that they do not present typical (re)design solutions, but patterns of process and project plans.

Proposed reengineering patterns are “Componentizing by building a facade”, “Changing interfaces in a client-friendly way” [Stevens98], “Type check elimination in clients”, and “Architectural extraction using prototyping” [FAMOOS98].

Besides the always ongoing topic of modernizing old systems to the latest buzz-word technology, there are four main areas of research in reengineering:

- Reverse engineering / Design recovery
- Program understanding
- Restructuring / transformation
- Methodologies for reengineering

Reverse engineering / design recovery

Reverse engineering is the process of analysing a computer system to create representations of it on a higher level of abstraction. Because this in some way is opposite of normal software development, it is called reverse engineering.

The principles of reverse engineering are clear enough in theory; group together constructs and replace them with a corresponding construct of higher abstraction. But this is not so simple in practice. Often one seeks to reconstruct a design from source code, and a good design includes the thoughts behind the decisions, but this is lost in the source code. In fact, Corbi stated that “automatically recapturing design from source code is considered infeasible” [Corbi90]. But despite such pessimistic predictions, a lot of work is being done in developing new approaches and programs for reverse engineering.

For example, Sneed and Jandrasics describes a tool that reverse engineers COBOL-74 source code into a higher level functional description in their paper “Software Recycling” [Sneed87].

REFINE, by Reasoning Systems Inc., is a reengineering package that can create object oriented databases from source code in a range of languages. Queries can then be made to the database to produce different views on the system [Kozaczynski92].

The Maintainer’s Apprentice is a toolset that can produce specifications from assembly language code. The specifications made by the program are functionally oriented, with a syntax similar to Pascal. It was made by REFORM, a project involving IBM, in the beginning of the 1990s. Lano and Haughton [Lano94] lists a full dozen of other tools for reverse engineering and design recovery.

Design recovery is the process of creating design descriptions for a system. This often includes reverse engineering of source code to design descriptions. To create good descriptions of the system, one also needs other sources of information, such as documentation and domain experts [Biggerstaff89]. The (human) domain expert is of course the most valuable source, since he/she has both formal and informal knowledge about the domain and the system. A domain expert knows the reasons for many development decisions, whereas a reverse engineering tool (at best) only can recognize these decisions.

Program understanding

In the area of program understanding researchers try to find out what programmers do to understand programs, how they should do it, and how computers can understand programs. Clearly, this borders on the area of reverse engineering, but the program understanding community is generally more focused towards people, and incorporate sciences such as congnitive psychology in their work.

According to Corbi, there are three ways of learning about a program [Corbi90]:

- Read about it (from documentation)
- Read it (static analysis, source code reading)
- Run it (dynamic analysis)

Much of the effort in this area has been aimed at determining how programmers understand a program from source code. There are three somewhat conflicting theories on the matter:

- The “bottom up” theory
  By reading the code, the programmer recognizes solution patterns and creates abstractions of the program as he/she goes. First, low-level patterns, such as “repeat until” or “step through an array” are recognized. Then, these patterns are put together to form higher-level algorithms, for example “bubble sort” or “quick sort”.
- The “top down” theory
  First, the programmer is told (or reads) what kind of a program is at hand, for example a complier. Then the programmer identifies modules and sub-modules in the program, such as the parser and the lexical analyser. Later he/she reads the code more closely and identifies the algorithms used in the different parts.
- The opportunistic theory
  The programmer does some bottom up, some top down recognition of the program, as different clues present themselves. The programmer searches for structures matching his/her general idea of the program, as well as making abstractions of low-level patterns.

To understand programs in a bottom up manner is difficult if the solution patterns are scattered around in the code [Letovsky86]. If different solution patterns are intertwined, or the program flow is not very sequential with respect to the source code, the patterns are split up, and the result is incomprehensible “spaghetti code.” Programmers tend to read the source code sequentially, and if there are no near-by explanation of the statements, they guess about their meaning.

Restructuring / transformation

Program Transformation deals with turning one description of a system (usually source code, but design and specifications are also possible) into another. Most often, the result is source code of a different language with improved characteristics. Program transformation has roots in the school of automatic programming, which has the vision of automatically deriving design from specifications, and implementation from design.

Restructuring is transformation of a description to make it easier to understand or less susceptible of errors when future changes are made [Arnold89].

Program translation is a term often used as a synonym to program transformation. However, program translation is normally used to describe a transformation from source code in one language to another language at the same level of abstraction. This makes program translation a subgroup of program transformation.

A program transformation can be defined as a relation between two program schemes. The transformation is said to be correct (or valid) if a certain semantic relation holds between the schemes [Hallstein89]. Usually, the semantic relation is equivalence, i.e. the semantic meaning of the scheme is preserved by the transformation.

William Chu [Chu93] divides program translation into two categories: Translation via Transliteration and Refinement (TTR) and Translation via Abstraction and Reimplementation (TAR). When doing TTR one just translates the program word by word and then does some optimizations on the result. TAR, on the other hand, creates an abstraction from the input program (reverse engineering). Optimizations are done on the abstraction before a new program is made by implementing the abstraction.

Program transformation is described more thoroughly in chapter 2.

Methodologies for reengineering

Many different persons have presented methodologies for reengineering. Examples include Brodie and Stonebraker [Brodie95], the RENAISSANSE project [RENNAISANSE98], the DOMIS project and the REDO project.

The DOMIS (Distributed Object Management Integration System) project was run by the MITRE corporation, and reengineered a large US Air Force computer system. They also published a methodology for modernizing such systems [DOMIS96]. The methodology focuses on remodularization, object orientation, and language independent communication between modules with CORBA.

The REDO methodology for reengineering describes six stages in the process [Zuylen93]:

1. Assess existing state
2. Install application in reengineering environment
3. Reverse engineer
4. Establish test procedure
5. Reengineer / Reimplement
6. Handover

However, none of these methodologies have been adopted by the reengineering community in general. This is quite different than in the software development community, where methodologies are immensely popular (although they tend to be replaced every few years.)

Stevens and Pooley [Stevens98] have tried to explain the lack of success the reengineering methodologies have had. The prime reason, they argue, is that reengineering projects are inherently more vulnerable to social and political problems than software development projects. There might be considerable resistance to drastically changing legacy systems, because it upsets the status quo. A reengineering of a system may also turn into a reengineering of work processes, with all the political dangers that area possesses. Software engineering research does not usually deal with such problems, and has difficulty with incorporating social/political issues into a methodology. On the other hand, because reengineering projects often fail for political reasons, it is difficult to evaluate the applicability of a methodology with normal software engineering methods.

Besides from social and political problems, organizations and their projects differ very widely. The size and nature of reengineering projects are so different that a methodology must be very general to cover them all. This could easily make the methodology too general for practical use.

If a methodology describes details, it must either restrict itself to a specific domain, and with that restrict its market, or be huge. A methodology that covered “everything” in detail would be so large that nobody would bother to read it.

This problem also exists in the software development field, but there has still been developed methodologies that have mastered the balance between generality and detail reasonably well. This leaves hope that a decent methodology might appear when the reengineering field matures.

Categories of reengineering projects

Of course, every reengineering project is unique, and no schema of categories can expect to match all projects to an appropriate label. During my work on this thesis, I have not found any classifications of different reengineering projects. So, for the convenience of the discussion later in this thesis, I will divide reengineering projects into three categories, based on which system artifacts are used and produced by the reengineering project. The three categories are porting / recycling, renovation, and redevelopment.

Porting / recycling

A porting project is a project that moves a system from one language or hardware platform to another. Porting projects need not include reengineering, often it is just a rework of the source code to make it compile on a different hardware configuration. When porting projects do include reengineering, it is often called software recycling, as otherwise unusable systems are reintroduced to the organization. (See the dictionary, appendix A for a more formal definition of software recycling.)

Porting or recycling projects do little else than program translation via abstraction and reimplementation (TAR) [Chu93]. Source code is reverse engineered to some design description or just an abstract syntax tree (AST). Then new source code is generated from the abstraction. The code may be restructured to improve maintainability or efficiency, but no functionality is altered. This corresponds to Sneed’s definition of reengineering: Zero percent of a program’s functionality is altered or enhanced [Sneed95].

A typical porting project is presented by Boyle and Muralidharan in their article “Program Reusability through Program Transformation” [Boyle84]. A LISP program had to be ported to FORTRAN to be of use to the general public. The program used a high degree of recursion, which is impossible to do in FORTRAN. A transformation program was used to abstract the program’s logic and rewrite it with legal FORTRAN techniques.

Renovation

Renovation projects are projects that do some reverse engineering on a system, and then gives it an overhaul before it is reimplemented. Source code and other sources of information are first used in a design recovery phase, where the structure of the old system is revealed. Then the design is modified to fit a new set of requirements. From here on the project is like a normal software development project, except that the developer possibly has a lot of recovered design and possibly can generate large parts of the implementation automatically.

Most reengineering projects fall into this category. A typical example is the DOMIS project [DOMIS96], which both remodularized a large system, reimplemented some of the modules, and added new interfaces for the users.

Redevelopment

This is the more exotic category of the three. In a redevelopment project, one discards the current implementation completely, and creates a new one from the specification or design.

This could be the case if the organization really wanted a recycling project, but decided that reverse engineering was infeasible for technical or economical reasons. However, it is more plausible that the organization has found the current implementation too messed up for salvaging, or that the design will be changed so much that all the code would have to be rewritten anyway.

The crucial point for this type of project is that the specification (or design) must be 100 percent reliable. Very often, these high-level system descriptions have not been updated during maintenance, so that the source code is the only authoritative description of the system.

One concrete example of this type of project is the TTM3 (Time To Market 3 months) project, which was conducted in Ericsson in 1998. The project team set out to recreate the subscriber services part of the AXE telephone exchange with a completely new architecture, and had to throw away all the old source code.

CHAPTER 3

THE BASICS OF PROGRAM TRANSFORMATION

General

In principle, program transformation is the process of rewriting one system description into another using a set of transformational rules. (See dictionary for a more formal definition.) A transformational rule has a left-side pattern and a right-side pattern, and possibly some conditions. The patterns can be written as regular expressions. When the left-side pattern matches a string in the source program and the conditions are satisfied, the string is substituted with the instantiated right-side pattern.

The most common and well-known form of program transformation is compilation. In compilation, a program written in a high-level language (source code) is transformed into a program in a low-level language (assembler or machine code). The compiler does this by applying a set of transformational rules that replaces each high-level statement with one or more low-level instructions. Optimizing compilers (almost all compilers these days) also go through a step of fine-tuning the produced code to run faster. This is also accomplished by program transformation. A set of rules recognize inefficient constructs and replace them with more efficient solutions.

This shows that program transformation can move a program from one level of abstraction to another (compilation), or keep it at the same level (optimization). As a transformation between levels of abstraction can move upwards, to a more abstract level, or downwards, to a less abstract level, we get three forms of transformation:

1 Downwards transformation

2 Upwards transformation

3 Optimizing (horizontal) transformation

Downwards transformation

In compilation, a program in a third generation language is turned into assembler code (second generation language), and then to machine code. Assembler code is, in principle, on the exact same level of abstraction as machine code, the instructions are just written in a human-readable form instead of zeros and ones. As an example of compilation, let us look at a piece of C code and the corresponding MIPS assembler code[1]:

C code: `earnings = (income1 + income2) - (expence1 + expence2);`

The variables are assigned to registers \$10, \$11, \$12, \$13, and \$14, respectively. \$8 and \$9 are used as temporary variables.

MIPS code:
`add $8,$11,$12` (temp1 = income1 + income2)
`add $9,$13,$14` (temp2 = expence1 + expence2)
`sub $10,$8,$9` (earnings = temp1 - temp2)

Because the MIPS instructions only have three operands, the evaluation has to be done in three steps. The transformational challenge is to rewrite the expression into expressions with three operands, and then translate the expression to MIPS assembler code. In the compilation information

1.The examples in this section are inspired by the examples in Patterson & Henessy’s “Computer Organization and Design” [Patterson94]. The first example is taken directly from the book, the others are adjusted to fit the text.

such as variable names and comments are lost, but normally this does not matter, as no one will read the compiled code.

Upwards transformation

Program transformation to a more abstract level is the essence of reverse engineering. Machine code or assembler code is transformed into third-generation language, or third generation language is transformed into design descriptions. But reverse engineering faces a fundamental problem: Meaningful information lost in the downwards transformation has to be recreated to add value to the descriptions. Let us see what a decompiler might have made of the MIPS code from the previous example:

MIPS code:
```
add $8,$11,$12
add $9,$13,$14
sub $10,$8,$9
```

C code:
```
v008 = v011 + v012;
v009 = v013 + v014;
v010 = v008 - v009;
```

The decompiler has not really lifted the program to a higher level of abstraction, it has really just written assembler code in the C language. It would take an optimizing step to turn the code into one statement:
```
v010 = (v011 + v012) - (v013 + v014);
```
Still, there are no clues as to what role the piece of code plays in the program. That would have to be answered by documentation or system experts.

Optimizing (horizontal) transformation

Optimizing transformation leaves the transformed program at the same level of abstraction. Depending on the purpose of the transformation, this activity can also be called restructuring or pretty printing. There can be many reasons for doing the transformation, see “Impact on quality” on page 35.

For efficiency reasons, the previously shown MIPS code can be rewritten to avoid the use of two temporary registers:

MIPS code:
`add $10,$11,$12` (earnings = income1 + income2)
`sub $10,$10,$13` (earnings = earnings - expence1)
`sub $10,$10,$14` (earnings = earnings - expence2)

This requires the transformational system to recognize that the registers $8 and $9 (and the parentheses in the C code) was unnecessary, since the order of the operations are unimportant.

For the sake of readability and maintainability, a program may be restructured to use FOR loops instead of GOTOs. Consider for example this C function:

```
void fill_matrix(matrix a, int m, int n)
{
 int   i, j;
 i = 0;
 Mloop:  j = 0;
   Nloop:  a[i][j] = rand() % (2 * N + 1) - N;
   j++;
   if(j <= n) goto Nloop;
 i++;
 if(i <= m) goto Mloop;
}
```

A transformational rule can recognize what elements constitutes a loop, and create a FOR statement to do the same:

```
void fill_matrix(matrix a, int m, int n)
{
 int   i, j;
 for (i = 0; i < m; i++)
 {
   for (j = 0; j < n; j++)
   { a[i][j] = rand() % (2 * N + 1) - N;
   }
 }
}
```

The four levels of program transformation

Kozaczynski, Ning, and Engberts [Kozaczynski92] divided program transformation into four levels, depending on how deep an "understanding" the transformation system has of the program.

Level 1: Text-Level Transformation

When the transformation system sees the program as nothing but character strings, string matching can be used to identify elements that conforms to the left-side of the rules. This makes the transformation operate similar to the search-and-replace function found in most text editors. For example, we could wish to rename the variable `i` in the C function in the previous section to `mIndex`. An automatic search-and-replace would produce this result:

```
vomIndexd fmIndexll_matrmIndexx(matrmIndexx a, mIndexnt m,
mIndexnt n)
{
 mIndexnt   mIndex, j;
 for (mIndex = 0; mIndex < m; mIndex++)
 {
   for (j = 0; j < n; j++)
   { a[mIndex][j] = rand() % (2 * N + 1) - N;
   }
 }
}
```

Obviously, more understanding of the environment surrounding an '`i`' is necessary to get what we want.

Level 2: Syntactic-Level Transformation

Syntactic information about a program can be obtained by parsing it into an abstract syntax tree (AST). The parsing into the AST is itself a form of upwards transformation, as the program text is replaced by syntactic elements like variables, operators, and statements. On this level, the left-side of the rules are abstract syntax patterns instead of string patterns, which simplifies the situation a great deal.

REFINE is a transformational system which works on ASTs. If provided with the definition of a language, REFINE recognizes variables, operators and statements. To rename the variable `i` to `mIndex`, we could have issued this command to REFINE:

```
IF $i -> $mIndex
```

Level 3: Semantic-Level Transformation

Semantic-level transformation is also performed on an AST, but the AST is enriched with relations between the elements, to represent the semantic relations between the syntactic elements of the program.
For example, the AST for the C function could be supplied with information about which statements belonged to which FOR-loops, and which variables were declared inside which functions. Then it would be possible to rename i to Mindex just in the fillmatrix function:

```
IF $i ->T $mIndex
where T= statement is-element-in fillmatrix
```

The where-statement is a says that the transformation T should only be performed on variables named '`i`' in statements inside the fillmatrix function. According to Kozaczynski, semantic-level information is sufficient to support correctness-preserving transformations such as code optimization and restructuring [Kozaczynski92].

Level 4: Concept-Level Transformation

Transformations on this level also incorporates knowledge of higher-level concepts in the rules. These concepts can be programming concepts, architectural concepts or domain concepts. For example, a rule can specify that only IF-statements that are part of a routine sorting customer addresses shall be transformed.

Recognizing programming concepts is not so difficult, compared to the other concepts, since a programming language has a complete specification. A parser for the language can be generated from a model and grammar of the language.

Architectural and domain concepts are more abstract, and by nature more difficult to recognize. According to Kozaczynski, two fundamental questions must be addressed: What to recognize, and how to recognize it.

A detailed domain model and a corresponding architectural model will help a great deal with the first question. These models should specify a concept classification hierarchy, but the models will probably never be as complete as a programming language model.

The how question may not be answered with only a syntactic approach, since the concepts may not be localized. Abstract concepts are more closely connected by semantic relations such as control flow, data flow and calling relations. Kozaczynski et. al. propose to recognize concepts by patterns of semantic relations in the AST and by defining a hierarchy of sub-concepts.

Transformational development

The basic idea of transformational development is to derive an implementation from the specification of the system through a series of downward transformations. Transformational development is related to automatic programming, but I will not use that term here, as it has a too general and vague meaning.[2]

If the transformations are sufficiently formally defined, each step of the development can be (relatively) easily verified. To make transformations operate in small, manageable steps, the language used should facilitate system descriptions at many levels of abstraction. (Considerably more than the four levels defined in the dictionary for this paper). Such languages are commonly called wide spectrum languages.

2.The term “automatic programming” has a likeness to the end of the rainbow: it always seems pretty close, but when you get there it has moved some place else. Belzer states that “at any point in time, the term has usually been reserved for optimizations which are just beyond the current state of the art” [Belzer85].

Hallsteinsen et. al. [Hallsteinsen89] describes an approach for transformational development with SDL and CHILL (CCITT HIgh Level Language). They argue that SDL and CHILL are such wide spectrum languages, well suited for this sort of development. Their idea is to first describe the system on a fairly abstract level in SDL, then apply a set of transformational rules to make the SDL more implementation oriented. In the last stages the SDL is transformed to high-level CHILL and then to low-level CHILL code.

Problems with restructuring / Program transformation

Upwards and optimizing transformation have some common problems: the program transformed by a computer should still be readable for a human. In a very interesting article, Frank Calliss pointed out the largest problems with automatic code restructurers [Calliss88]:

- Understandable, but non-structured code is transformed into structured, but incomprehensible code
- Auto-created variables get meaningless names
- Comments are detached from their code and become meaningless

Although this article was written to explain the damage automatic restructuring can do to COBOL code (as a response to Miller and Strauss' article about the benefits of restructuring COBOL [Miller87]), it is a good description of the problems with restructuring in general.

The rules that ensures structured (and thereby readable) code in general, might not be the best in every case. Calliss uses simulated CASE statements (COBOL did not have CASE statements before COBOL85) and loops with mid-exit or multiple exits as example.

If an automatic restructurer changes all unstructured code into the three allowed structures[3], the logic in these constructs will be cluttered, not clarified.

When new names are introduced, the computer has (obviously) no chance of giving these very meaningful names, as mentioned in "Upwards transformation", page 27.

If code is moved or split, there is a great chance that comments in the code will become meaningless. Comments describing the purpose of a larger part of the program can be hidden away in a subroutine together with its surrounding code. Comments explaining specific operations will become quite puzzling if that statement is transformed into something equivalent, but in a different form. When code is split, comments can be moved with the wrong part. (After all, some comments describe the code above in the program, some the code below).

Better programming or Artificial Intelligence could probably improve the computers performance somewhat. After all, the chip manufacturers have managed to produce processors that can guess the outcome of a test 90% of the time, so that it can process instructions ahead of time (Intel's Pentium II). But I suspect that these problems can only be solved by humans, since they require judging the quality of names and solutions. (More on measuring quality with computers on page 50.)

3.All programming logic can be expressed with three structures: Sequence, selection (such as IF-THEN), and iteration (such as FOR-loops). For example, a CASE statement can be expressed as a series of IF-THEN statements, and FOR, WHILE, and UNTIL loops are interchangeable, since they are essentially the same. This was probably first stated by Böhm and Jaccopini [Böhm66].

CHAPTER 4

RESTRUCTURING SDL

As the practical part of my master's thesis, I decided to make a program to restructure SDL diagrams for better readability. The program was first tested on a small example, created to fit the purpose, and later on a real world example, a diagram from the T.30 project (see "The Test Case" on page 71). The program was called `transformer.pl`, since it would be a transformational system.

Impact on quality

Many aspects of a program's quality can be improved by restructuring. In his book "Software Evolution: the Maintenance Challenge", Lowell J. Arthur [Arthur88] lists 11 qualities that restructuring may improve:

- Maintainability
- Flexibility
- Reliability
- Reusability
- Usability
- Efficiency
- Testability
- Integrity

- Portability
- Interoperability
- Correctness

The ISO standard for software quality [ISO92] divides maintainability into four characteristics: Analysability, changeability, stability and testability. Source code that is difficult to read and comprehend has a low degree of analysability. Poor analysability will usually lead to poor changeability.

The same ISO standard divides usability into three characteristics: understandability, learnability, and operability. Source code with poor readability will typically have a low degree of understandability. Poor understandability will probably lead to poor learnability. This means that the readability is an integral part of a piece of software's maintainability and usability.

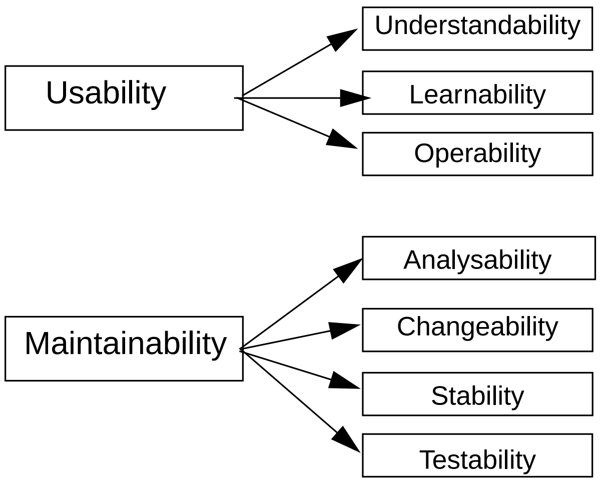


Figure 4.1 Part of the ISO software quality model

SDL is a neat and tidy language, compared to C, with its many low level commands, or Perl, with its infinite possibilites for obscure and incomprehensible code. Still there are plenty of opportunities to produce complex and unintelligible SDL descriptions.

Restructuring vs. pretty printing

Pretty printing is closely related to restructuring done for readability purposes. The goal is the same: to make the program easier to read and comprehend. It is possible to argue that readability is easiest to improve with pretty printing, and that using program transformation for this is a bit of an overkill.

I think that this is not the case. A pretty printer changes only the layout of the program (typically indentation and fonts), while a restructurer can change the syntax or semantics of the program to improve the readability. With restructuring, it is possible to change the program much more deeply than with a pretty printer, and thereby (hopefully) achieve better results

An introduction to SDL

This section is not intended to give the reader a deep understanding of SDL, but merely to present the parts of SDL mentioned in this thesis. For a more complete coverage of SDL, I recommend the books “Systems Engineering Using SDL-92”, by Olsen et. al. [Olsen94], or “SDL. Formal Object-Oriented Language for Communicating Systems”, by Ellsberger, Hogrefe, and Sarma [Ellsberger97].

SDL stands for Specification and Description Language, and is defined by ITU in the standard Z.100 [ITU93]. SDL has evolved from state diagrams in the 70s, and is now a full-fledged programming language with variables, signals, procedures, types, and inheritance. SDL is a graphical language, but there is also defined a textual representation of SDL called PR, Prose Representation.

The basic element of a SDL system is the **process**. The process has **states**, behaviour, and can send and receive **signals**, much like a system process in a computer. The signals are defined to be asynchronous, so it is not possible to know when a signal will arrive. Different SDL processes operate concurrently, so SDL is well suited to model real-time systems.

Figure 4.2 and Figure 4.3 show the behaviour of a washing machine, described with a informal state diagram and a SDL process.

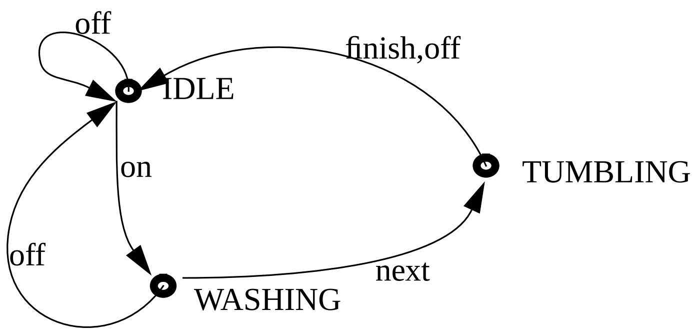


Figure 4.2 Washing machine, informal state diagram

The empty oval symbol at the top left of Figure 4.3 is the **start symbol**. This symbol defines the initialization of the process. The semi-oval symbol beneath it marked “IDLE” is a **state symbol**. A process “rests” in a state, and when a signal comes in, it spurs into action, may do some tasks, and then moves to another state (or back to the same one). The symbols marked “on” and “off” below the state symbol are **input symbols**. The symbols marked “ok” and “nok” are **output symbols**.

The semi-oval symbols below the input symbols are called **nextstate symbols**. They look exactly like state symbols, but defines which state the process should go to next. The dash in one of the nextstate symbols means that the process should remain in the same state, in this case IDLE.

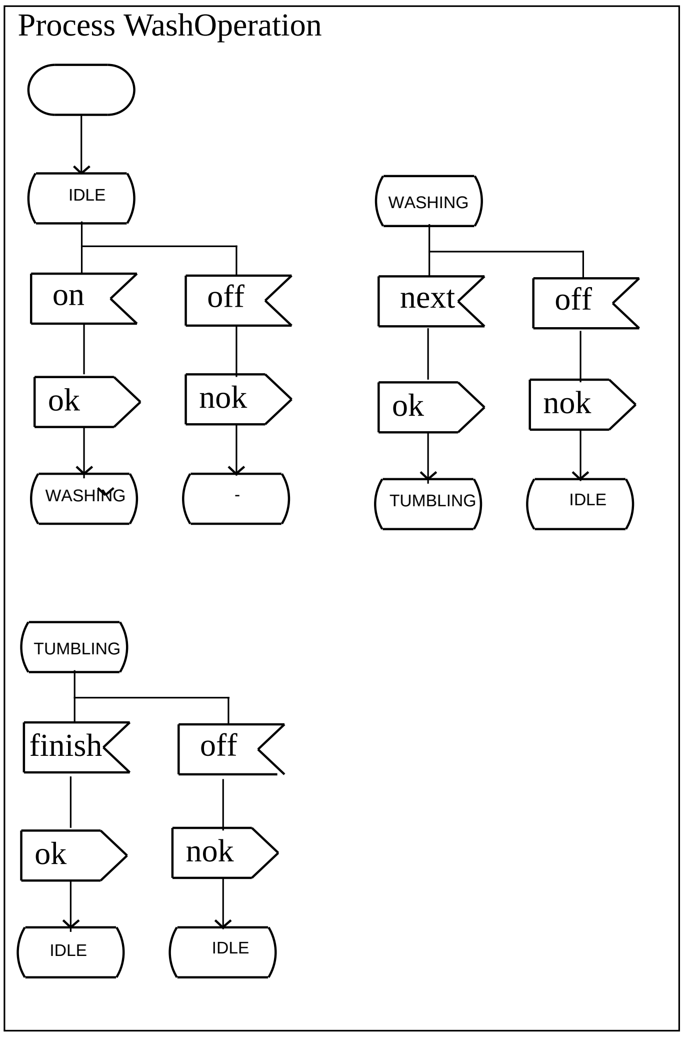


Figure 4.3 Washing machine, as an SDL process

For modularization of descriptions, SDL has the concepts of **procedures** and **connections.** Procedures are very similar to procedures in other programming languages. Connections are pretty much like GOTOs.

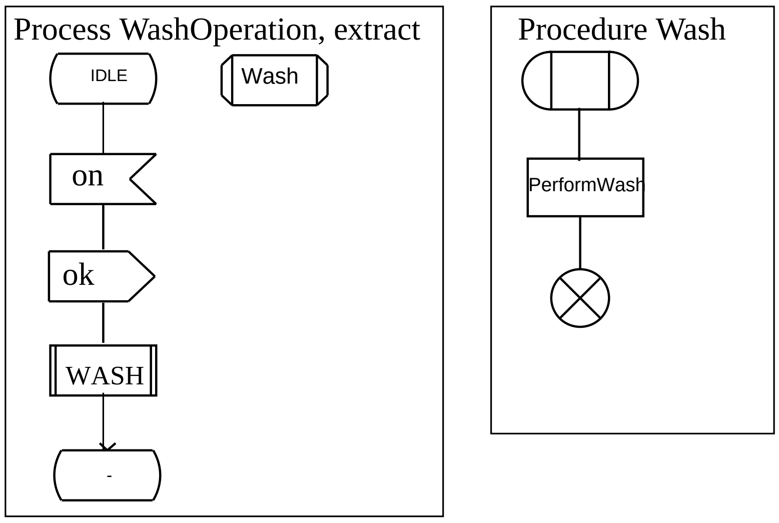


Figure 4.4 A process that uses a procedure

The rectangular symbol marked “WASH” in Process WashOperation in Figure 4.4 is the **procedure call** symbol. The odd-looking symbol in the right corer is the procedure reference symbol. This tells that the procedure is defined in another diagram.

The symbol in the upper right corner of Procedure Wash in Figure 4.4 is the **procedure start** symbol. The circular symbol with a cross is the **return symbol**.

Instead of using a procedure, the behaviour could be expressed with a connection, as shown in Figure 4.5.

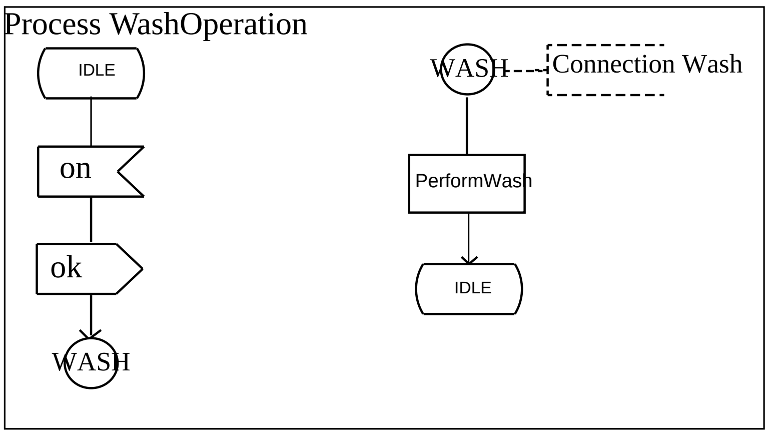


Figure 4.5 The same process, with a connection instead of a procedure

The circular symbol marked “WASH” to the left in Figure 4.5 is called a **join symbol**. Its effect is to send the process to the connection named “WASH”. The same symbol used on the right of the figure is called a **label symbol** in that context. The meaning of “join” and “label” corresponds pretty well to the meaning of “goto” and “label” in low-level languages. The dashed box to the right of the label symbol is a comment.

An SDL **system** is defined to consist of **blocks**, which in turn contains one or more processes. For processes in different blocks to be able to communicate, there must exist a **channel** between their parent blocks. Communication between processes in the same block are sent through **signalroutes**.

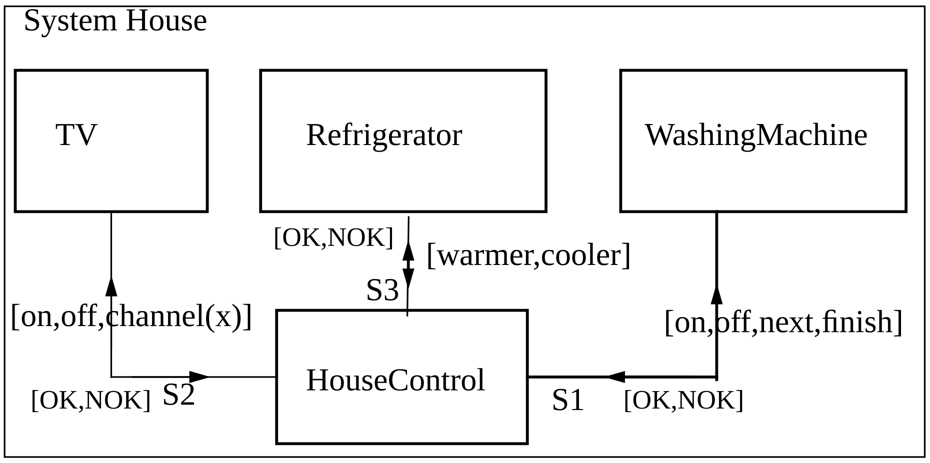


Figure 4.6 An SDL system with blocks and signalroutes

Figure 4.6 shows the system “House” with the blocks “HouseControl”, “TV”, “Refrigerator”, and “WashingMachine”. The arrows between the blocks are the channels, named “S1”, “S2”, and “S3”. The words enclosed in brackets close to the channels are the names of the signals that can be sent over the channel. For example, HouseControl can send the signals “on”, “off”, “next”, and “finish” to WashingMachine. WashingMachine may send either “ok” or “nok” in return.

Figure 4.7 shows the block WashingMachine. The channel S1 enters the block, and all incoming signals are sent to the only process in the block, WashOperation, depicted in the octagonal process symbol.

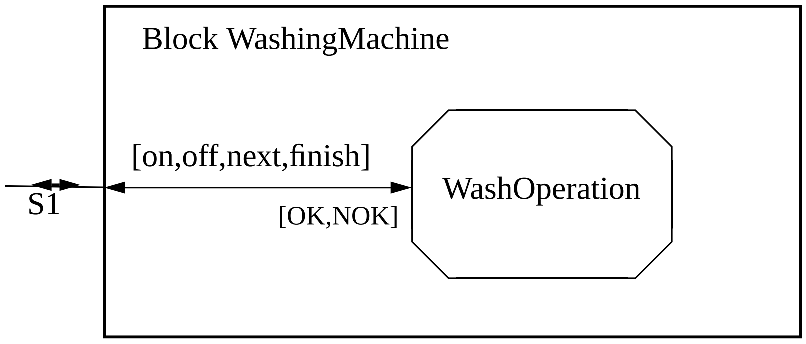


Figure 4.7 A block with one process

Development models

Transformation to improve readability can be used in several development models. For example, `Transformer.pl` could be used to improve SDL descriptions in a transformational development, reengineering process, or in software maintenance.

Transformational development

Transformational development derives an implementation from a specification through a series of transformations (see "Transformational development" on page 31). `Transformer.pl` could be used in one of these transformations to improve the readability of the SDL that is generated from the specification. In the test case (see page 71) `transformer.pl` was used in this way. A system description in SDL, automatically derived from the T.30 standard, was transformed with the program to create a more readable version.

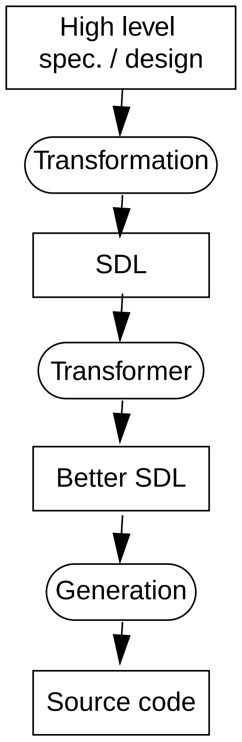


Figure 4.8 `Transformer.pl` in a transformational development process

Reengineering

In a SDL-oriented reengineering process, SDL will be generated from old source code by a reverse engineering tool. This SDL will most probably not be very readable. My program could reorganize the SDL to be more readable before the forward engineering began.

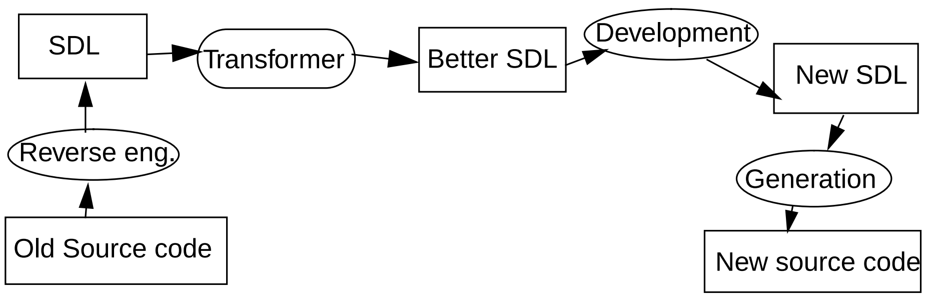


Figure 4.9 `Transformer.pl` in a reengineering process

Maintenance

As previously discussed, all software tend to degrade over time (see Lehman’s laws on page 13). `Transformer.pl` can be used to bring degraded SDL descriptions back to a specified format.

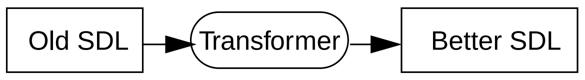


Figure 4.10 `Transformer.pl` in a maintenance process

Other efforts

The only other program that can reorganize graphical SDL descriptions is the SDT program package from Telelogic AB. The package has a “tidy up” function that reorganizes SDL diagrams. An example of the function’s work is shown in Figure 4.11.

The purpose is to enhance the readability for the programmer. The function works well, but clearly has not incorporated many rules to improve the SDL. The rules are not explicitly stated in the documentation, and from my testing I only discovered three:

1. Gather all declarations on the first page
2. Each state shall be described in a different flow.
(But there can be several flows on a page).
3. The states shall be listed alphabetically, and all connections shall be defined after the states.

The problem with the function is that it compresses the diagrams and leave little room for new additions. The SDT approach does tidy the diagrams up a bit, but breaks most of the rules for readable SDL set forth in this paper (see page 49).

SDT will place all nextstates 50 pixels below their predecessor, and thus disaligning the nextstates if they had been aligned. The program may also create new connections if a flowline does not fit on a page, instead of moving the flowline to another page.

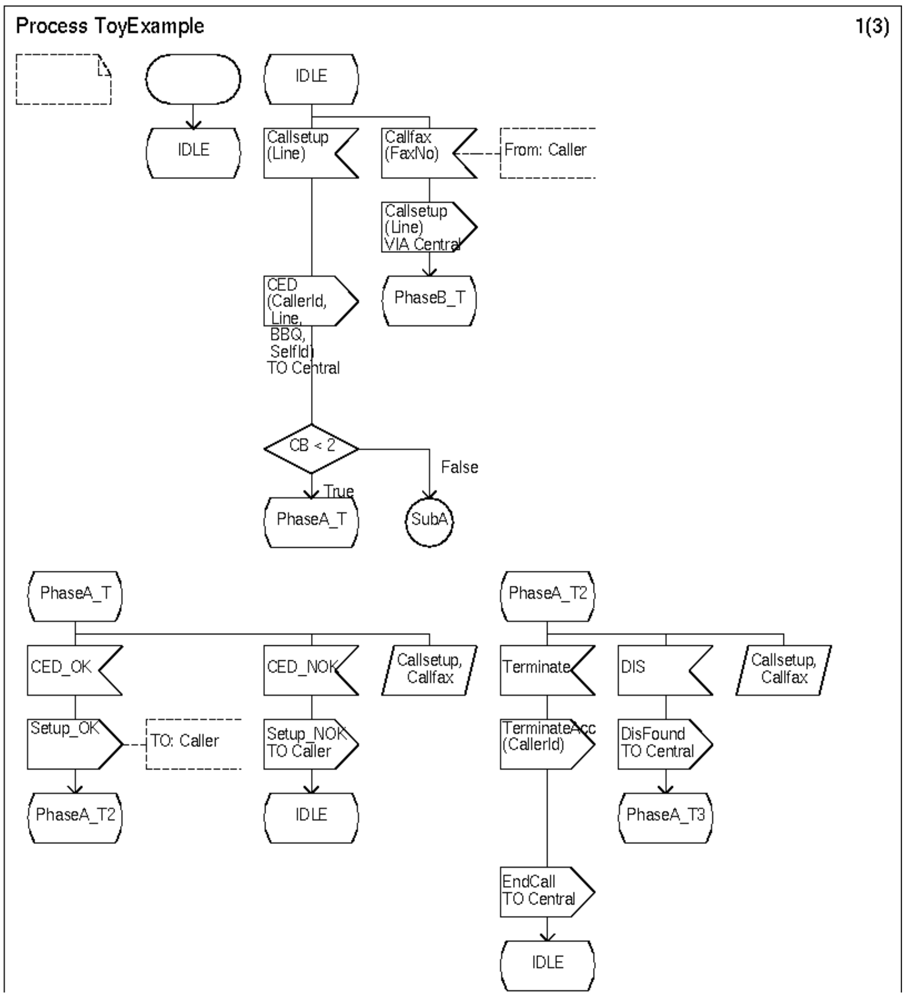


Figure 4.11 The process ToyExample after using the “tidy up” function. (The original Toy Example is in Appendix A.)

Principles

The goal of the restructuring was to make messy SDL code conform to a code standard that ensured good readability. However, there were few guidelines for readable or maintainable SDL diagrams in the literature. There are many proposed standards available for writing “good” code in most programming languages, but little has been written for graphical languages. A quick search on the Internet gave me guidelines for writing Java, C++ and Perl, but when I searched for “readable UML” or “pretty MSC”, I found nothing.

A fruitless trip to the bookstore revealed that the books on UML (I did not find any on MSC) showed little attention to readability. The authoritative UML book by Booch, Rumbaugh and Jacobson [Booch99] does have some tips on how to draw “structured” UML, but this is just a very minor topic.

The only source of rules for “good SDL” I found was Bræk & Haugen’s “Engineering real time systems” [Bræk93]. The book divides the rules into three classes: Analysis rules (A-rules), structural rules (S-rules), and notational rules (N-rules).

The analysis rules are meant to tell the designer how to analyse a system before the SDL is written. The structural rules issue guidelines for SDL modelling of a system. The notational rules are concrete advises to how the SDL should look on paper. Obviously, the S-rules and the N-rules are the most interesting for this thesis, as they directly influence the readability of SDL diagrams.

Unfortunately, most of the rules were so general that a computer would not be able to detect a violation of them, not to mention correcting such an error. For example, a N-rule named “Atleast, finalized” said:

“In specialising, take care to balance flexibility and analysability properly using ATLEAST and FINALIZED to constrain the virtual types”.

It is just too difficult for a computer to decide if virtual types are flexible “enough”.

I decided to divide the rules I would use into three categories: rules with correctable violations, rules with detectable violation and rules without detectable violations. Of the 65 rules presented in the book, only two fit into the first category, and another two in the second. In addition, I made some new rules after noticing some typical problems in the SDL diagrams I have been working on.

The rules I selected for the experiment were as follows:

Rules with correctable violations:

N-rule: State and nextstate

1. Each page of a diagram should have only one state and the state should be at the top of the page. (Example in Figure 5.2 - Figure 5.5).
2. All nextstates shall be aligned at the bottom of the page. (Example in Figure 5.2 & Figure 5.3).
3. New: Connections should also be described on separate pages.

N-rule: Source and destination
Specify the sender of an input signal with a From: statement in a comment symbol. Specify the receiver of an output signal as part of the send statement or in a comment symbol. (Example in Figure 5.8 & Figure 5.9).

New rule: Signal parameters
If there are more than two parameters to a signal (one if there is a VIA or TO statement), the parameters should be listed in a text extension symbol to the right of the signal symbol. (Example in Figure 5.6 & Figure 5.7).

New rule: Decision layout
If decisions are used, the first alternative outcome shall be directly below the decision symbol, with the other alternatives spread out to the right. (Example in figure Figure 5.10 & Figure 5.11).

Rules with detectable violations:

N-rule: Names in macros
Each macro call shall have a unique number.

S-rule: Control flow
Branch on signals, not decisions.

New rule: Meaningful names
All names of variables and signals should be meaningful, preferably with a prefix and a descriptive identifier.

New rule: Connections
Connections should be used as seldom as possible. If a connection represents a recurring solution, use a procedure instead. If a connection just represents accidental reuse of behaviour, expand it on the places it occurs.

The program was to use a transformational approach in the restructuring. That meant that the changes made to the SDL should be described as transformational rules. The program should also follow the principle of correctness-preserving described by Boyle & Muralidharan [Boyle84]: The order in which the transformations are applied must make no difference, or we must control the order in which we apply them.

Non-detectable violations

Most of the rules by Bræk & Haugen are guidelines to help the developers create systems of high quality. For the program trying to verify if the rules are fulfilled, two problems arise. Firstly, even though a rule may have been followed, the results may be very difficult to distinguish from SDL produced without following the rules. Secondly, quality is inherently difficult to measure with a computer.

According to the transcendental view of philosophy, quality is something that can be recognized, but not defined. The quality of an item depends greatly of the context it is in, and is subject to personal opinion. Since quality cannot be defined, it is difficult for humans to communicate about quality. Computerized recognition is totally dependent on clear definitions, so all automatic evaluation of quality will be impossible, according to this line of reasoning. To get anywhere with computers in this matter, we have to abandon the transcendental view of quality and adopt a more pragmatic view.

To make the computer useful in measuring quality, we must assume that there is a connection between quality and some measurable characteristics of a SDL description. If the (easily recognizable) characteristics are present, the program must have the quality in question. The challenge is to identify these characteristics so that a computer program may recognize them. The problem with most of the rules is that they are qualitative, and have no easily definable characteristics. For example, the rule "adaptable components" state:

Achieve adaptable components by introducing virtual types. Ensure that such types get proper general names. Balance the adaptability by using ATLEAST to limit the redefinability.

It is simply impossible to define precise characteristics for a "properly general name". One could try to define "balanced adaptability", by making rules for the number of ATLEASTs compared to the number of virtual types. Still, it is unlikely that the ratio of ATLEAST/Virtual types would have any correlation to how adaptable the components were.

Detectable vs. correctable violations

The correctable violations make up a subset of the detectable violations. All these violations have some characteristics that can tell that the violations are present, i.e. the rules tell us how things should **not** be. For example, it is easy to check if a process uses decisions, but it is anything but trivial to replace the decisions with something else.

The essential difference is that the rules for correctable violations also contain detailed (enough) instructions of how things **should** be. That means that a computer can set things right, without human input. For example it is fairy easy to make sure that an SDL description has one state on each page, and put source/destination statements all its signals.

These rules are really the only ones suited for automatic restructuring, but unfortunately there are very few of them.

CHAPTER 5

THE EXPERIMENT

Choice of methods

To make my program as generally usable as possible and to simplify the programming, I decided that it should work on SDL descriptions in PR/CIF (Common Interchange Format). Since Perl is a great language for text manipulation, and I have some experience programming with it, I chose to use Perl for the experiment.

These choices were based mainly on personal preferences, but the approach also proved quite portable. There are three major vendors of graphical SDL tools: Cinderella, Telelogic, and Verilog. I tested my program with Telelogic's SDT on UNIX (SUN Solaris) during the development. Later, the program was tested with Cinderella on MS windows and Verilog's Geode on UNIX, and both worked reasonably well with my program.

PR/CIF is a plaintext format, where the SDL is written in its textual form, with CIF commands that control the layout. The CIF commands are specified in the ITU standard Z.106 [ITU92].

A SDL description in PR/CIF can look something like this[1]:

```
/* CIF ProcessDiagram */
/* CIF Page State_IDLE (2650,1800) */
/* CIF Frame (0,0),(2650,1800) */
/* CIF CurrentPage State_IDLE */
/* CIF Start (250,100) */
start;
/* CIF Line (350,200),(350,250) */
/* CIF NextState (250,250) */
nextstate IDLE;
/* CIF State (250,250) */
state IDLE;
/* CIF Line (350,350),(350,400) */
/* CIF Input (250,400) Right */
input DefineSub
/* CIF TextExtension (500,400) Right */
/* CIF Line (500,450),(450,450) */
(SubId,
AntDig,
SNB)
/* CIF End TextExtension */
;
/* CIF Line (350,500),(350,550) */
/* CIF Output (250,550) Right */
```

Design of the program

The program was called `transformer.pl`, since it would be a transformational system. A graphical SDL tool would generate a PR/CIF file. `Transformer.pl` read this file and made a new and improved PR/CIF file, which was then imported back into the graphical tool. With Verilog's Geode, there was not even a export/import step, as Geode stores its diagrams in PR/CIF.

1.This PR/CIF is from the toy example shown in appendix B.

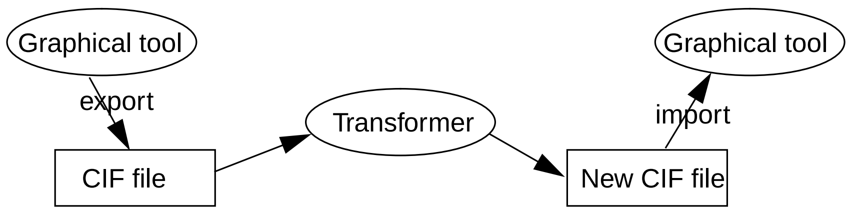


Figure 5.1 The interworking of transformer and a graphical SDL tool

The basic operation of the program was this:

1. Open a specified file and read it into an array[2] called `@sdlfile`.
2. Run a set of transformations on the array.
3. Write the array to a specified file.

The transformational rules were programmed as procedures, one for each rule. This made the general structure of the program very clean. The main part of the program looked like this (the code is from the final version, and have some procedures that are not explained until Chapter 6):

```
#Initialize: Open infile, read it into the sdlfile array.
print "Transformer starting\n";
open(INPUT,$infile) or
die("Could not open input file $infile.\n");
@sdlfile = <INPUT>;
close(INPUT);

#Perform transformations
if($insertComments)
{ &insertTextExtensions;
  &makeSignalTextExtension;
}

if($removeInvisibleJoins)
{ &removeInvisibleJoins;}
```

2.In Perl, arrays do not have a fixed length, so items can be appended and deleted from the array.

```
&expandConnections;
&stateOnNewPage;
&alignPages;
&resolvePageOverflow;
&alignNextstates;
&removeEmptyPages;

if($showWarnings)
{ &warnShortNames;
  &warnBranchOnDecision;
}

#Write transformed SDL to file
open(OUTPUT,">$outfile") or
die("Could not open output file $outfile.\n");
foreach $outline (@sdlfile)
{ print OUTPUT "$outline";
}
close(OUTPUT);
```

Implementation of the rules

Each rule was to be implemented as one procedure, but some of the more complex ones, like “state and nextstate”, was divided into multiple procedures for the sake of modularity. All the rules were in principle correctness-preserving, and the procedures should also be that. Therefore, the procedures had make sure that the file was still a correct PR/CIF description before terminating.

Since the method of recognition of candidates for transformation was string matching, the program should fall into Kozaczynski’s first level of transformation (see “The four levels of program transformation” on page 29). However, to be correctness-preserving, the program had to recognize some of the syntax and the semantics of the SDL description. I tried to accomplish this by setting careful conditions for when the rules should be applied.

State and nextstate

The rule was implemented as three procedures. `StateOnNewPage` would insert new pagebreaks, `alignPages` would align the symbols on newly created pages, and `alignNextStates` would align nextstate symbols.

If more than one state was defined on a page, extra pagebreaks was inserted. For example, a page with five states would be split into five different pages. Figure 5.2 shows the original page, Figure 5.3 - Figure 5.5 shows three of the five pages after the transformation.

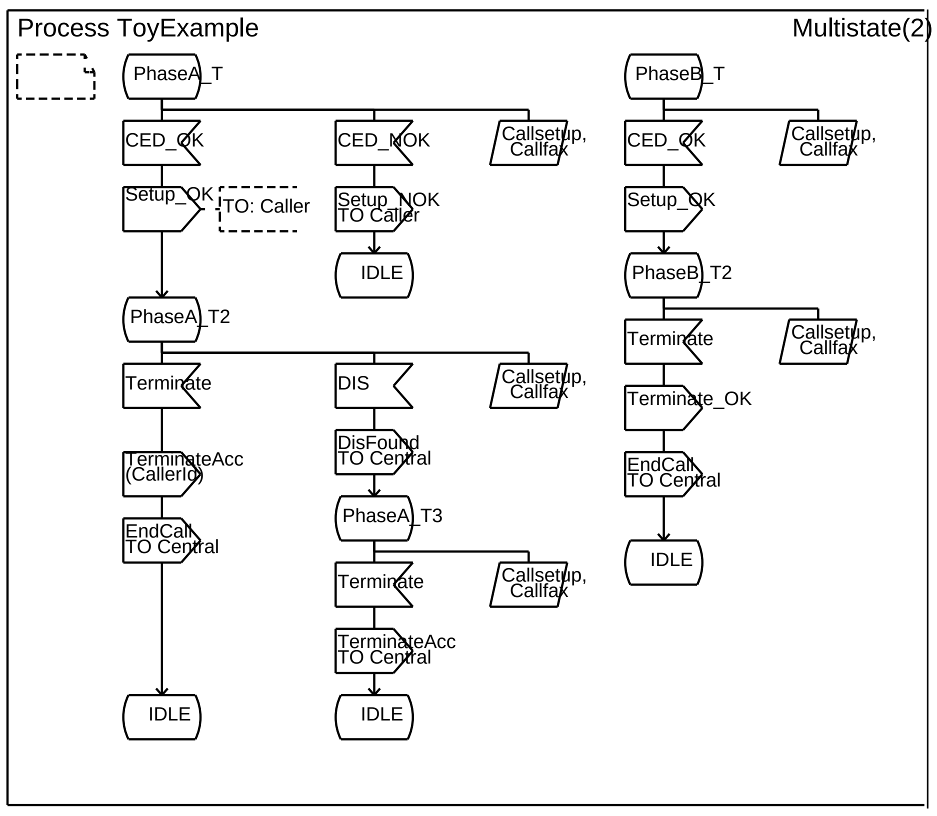


Figure 5.2 Five different states on one page, no alignment of nextstates

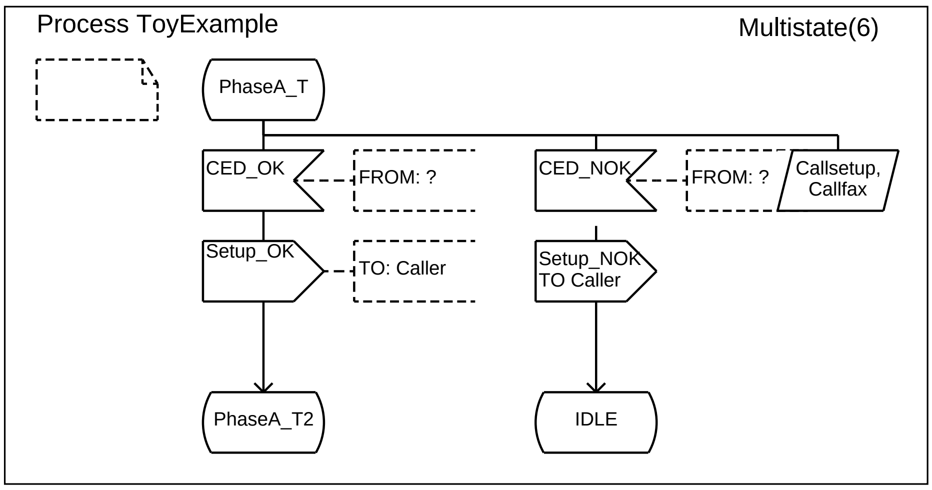


Figure 5.3 State PhaseA_T as the only remaining state on the page, nextstates aligned

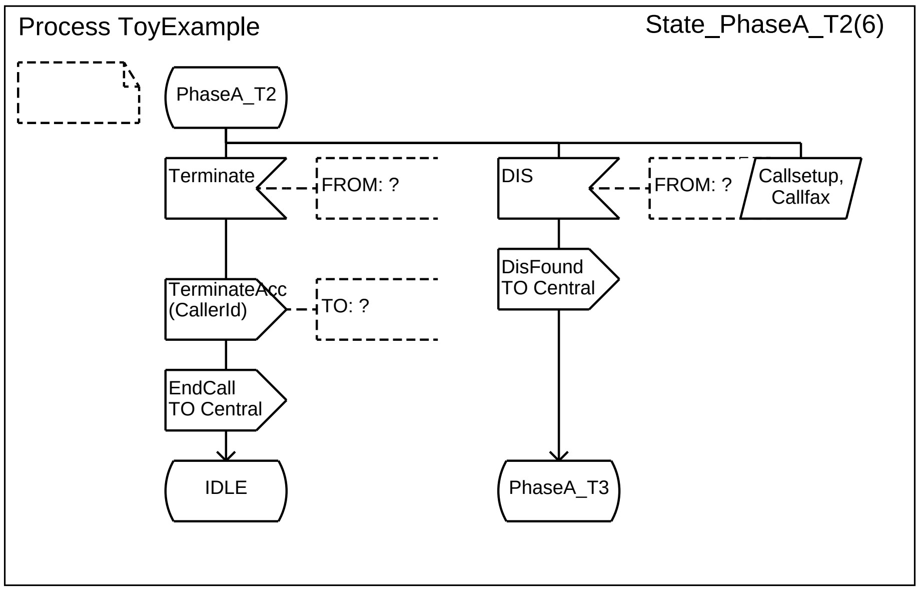


Figure 5.4 PhaseA_T2 on a separate page

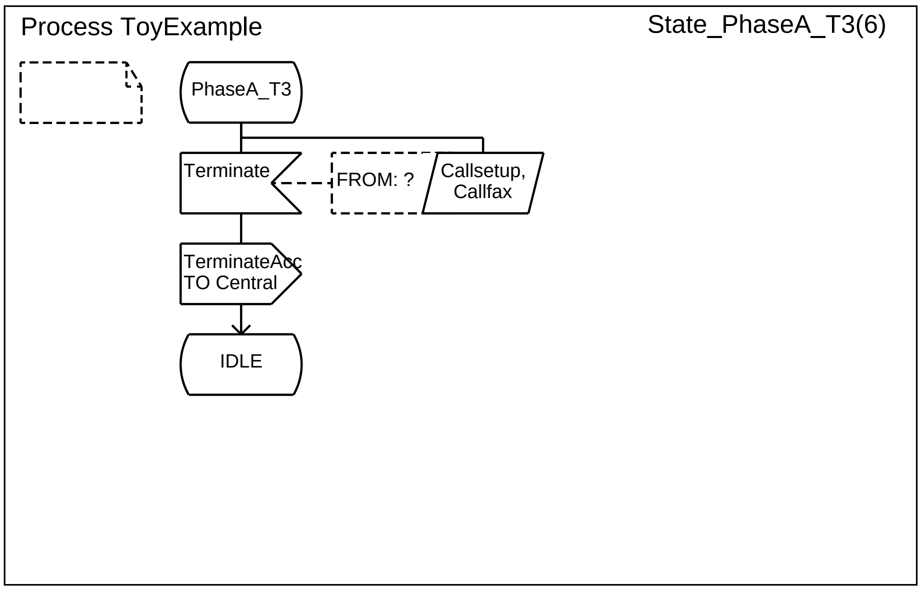


Figure 5.5 PhaseA_T3 on a separate page

Signal parameters

If a signal has too many parameters, the text will flow outside the symbol and clutter the diagram. The `makeSignalTextExtension` procedure created a text extension to the right of the symbol if the signal had three or more parameters. Figure 5.6 shows a signal with parameters that flows outside the symbol. Figure 5.7 shows the same signal with the parameters put in a text extension box.

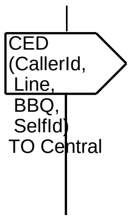


Figure 5.6 A signal with too many parameters

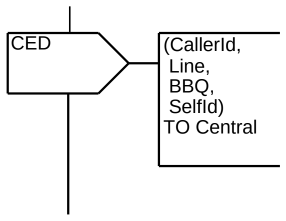


Figure 5.7 The signal, after creating a Text Extension box for the parameters

Source and destination

When the source or destination of a signal is specified, it is much easier for a casual reader to understand what goes on in a SDL system. However, it is not mandatory to specify sources and destinations in SDL, as this can be derived from the definitions of the channels and signalroutes. But these definitions are stored in the system and block diagrams, not the process or procedure diagrams, so it is not easily accessible to the reader.

Since the transformation program only looked at a process or procedure file, it had no way of knowing where a signal came from or was going to. Therefore, the program just added dummy text extensions and issued warnings to the user. Figure 5.8 and Figure 5.9 show how the dummy text extension is added to remind the programmer to fill in the source of the signal.

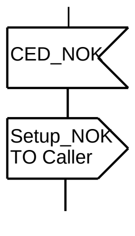


Figure 5.8 The signals CED_NOK and Setup_NOK, before applying the rule

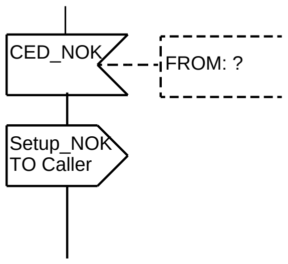


Figure 5.9 The signals after applying the rule

Decision layout

For lack of time, this rule was not implemented. On the test case, the decisions were manually fixed to follow this rule, since that made the rest of the layout easier to adjust. Figure 5.10 shows a typical example of undesired layout of a decision. Figure 5.11 shows a decision with the decision symbol to the left, and the alternatives in sorted order.

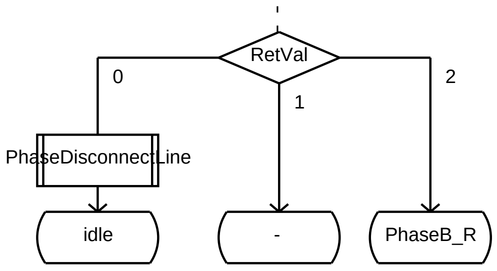


Figure 5.10 A decision that violates the rule

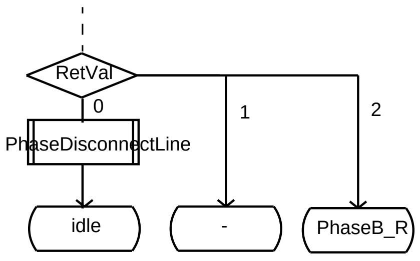


Figure 5.11 A decision that follows the rule

Control flow

According to the principles presented earlier, branching on decisions is undesirable. Unfortunately, there is no easy way to transform decisions into, for example, branching on signals. Because of that, the program had to just issue a warning on each decision. Of course, if decisions is used extensively in a diagram, the user will have lots of warnings scrolling across his screen, and be unable to read them all. The user might start ignoring all the output, and suffer from so called “warning blindness”. To avoid this, I included an option to turn off warnings.

Meaningful names

As discussed earlier, computers cannot recognize quality if it is not quantitatively defined. Obviously, the program could never turn cryptic names into meaningful ones, so it had to settle for issuing warnings about bad names instead. From the idea that short names often are not very meaningful, I decided that the procedure should warn if a name was shorter than 5 characters.

With these decisions made, the implementation was straightforward. The procedure `WarnShortNames` stepped through the array, and every time it found a DCL or SIGNAL statement with a name shorter than 5 characters it printed: “Warning: variable/signal *name-in-question* has a short name.”

Connections

Connections can be thought of as the GOTO of SDL. A join statement jumps to the definition of the connection, and there is no guarantee that the execution will return to the same place. The problems with GOTOs have been discussed for almost three decades, and I will not repeat them here. For a good explanation of these problems, see Miller and Strauss’ article “Implications of automatic restructuring of COBOL” [Miller87].

There are generally two ways to remove connections: write out the whole definition instead of each join statement, or turn the connection into a procedure and the joins into procedure calls. The expansion method is straightforward, but creates duplicate code, which is negative for maintenance.

The procedure approach facilitates code reuse and modularity, but has a call-return structure, as opposed to the GOTO-like jump structure of connection. The structure is easily transformed when the connection only returns to one place (all nextstates in the connection goes to the same state). Then all nextstate statements are made into return statements, and the joins are made into a procedure call and a nextstate statement.

When the connection can return to different states it gets a bit more complicated. One solution is to let the procedure return a value indicating which state the process should go to. Directly after the call statement the program must create a decision that branches on this value and then goes to the correct state. (Note that this would introduce a new violation of the decision rule).

The program gave the user three ways of removing connections:

1 Expand all occurrences of the connection
2 Expand only selected occurrences
3 Turn the connection into a procedure

For lack of time, transformation into procedures was only implemented for connections with single return points. Below is an example of a connection ‘SubA’, both expanded and transformed into a procedure.

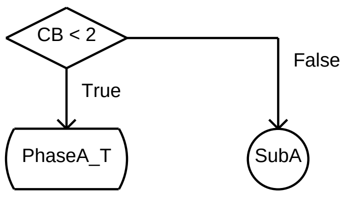


Figure 5.12 The join to the connection, before transformation

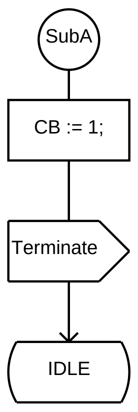


Figure 5.13 The definition of the connection subA

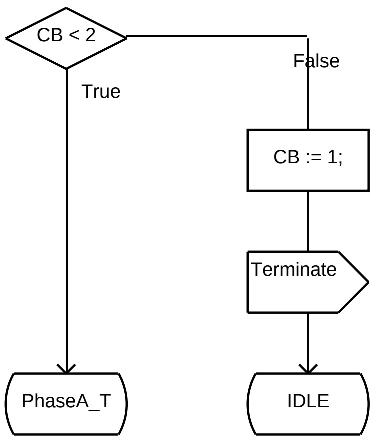


Figure 5.14 The connection expanded into the process

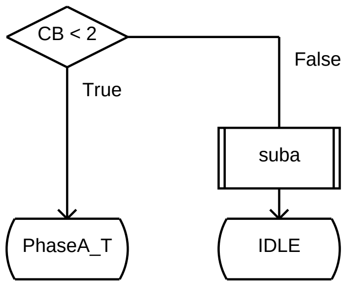


Figure 5.15 The join transformed into a procedure call and a nextstate symbol

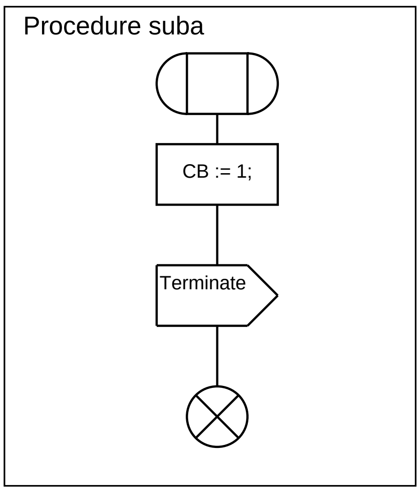


Figure 5.16 The generated procedure suba

Correctness of the rules

To feel safe that the rules does not destroy the semantics of a document, it should be proven that all the rules are semantic-preserving. For most of the rules, this is trivial, since they only alter CIF statements. In PR/CIF, the CIF statements are written as SDL comments, which means that they have no impact on the SDL semantics.

State/nextstate, source and destination, and signal parameters only change CIF statements, and must therefore be semantic-preserving.

Control flow and meaningful names does not alter the SDL at all, as they only issue warnings to the user. If fully implemented, however, the rules would have altered the semantics and would have needed to be proven correct.

The only rule that does change the semantics is connections. Expansion of connections, or transformation into a procedure do change the semantic of a SDL description.

An expansion of a connection results in equivalent semantics, since the join symbol is just shorthand for writing the whole connection. (See page 76, “Join”, in Z.100 [ITU93])

Replacing a connection with a procedure does indeed alter the semantics. However, when the transformation is described as an expansion and a procedure creation, the equivalence is clear. The steps in the transformation is as follows:

Precondition: All the nextstates in the connection goes to the same state.

1. Expand the connection. (See Figure 5.14.) Proven correctness preserving above.
2. Create a new procedure with all the expanded symbols, and change the nextstate in this procedure to a return symbol. (See figure Figure 5.16) This step does not alter the semantics of the process, as it just creates a procedure that is not used (yet).

3 Replace all the expanded symbols, except the nextstate, with a procedure call to the procedure created in step 2. Add a reference to the procedure. (See Figure 5.15.)

The procedure call, execution, return and nextstate results in the exact same behaviour as the expanded connection. The procedure call just transfers the execution to the procedure. (See page 81, “Procedure call”, in Z.100 [ITU93].) Since the procedure contains the same symbols as the expanded connection the execution will behave in the same way. The return symbol just transfers the execution back, to the first symbol below the procedure call. This is the nextstate symbol, pointing to the sam state as the ones in the expanded connection. (See page 71, “Return”, in Z.100 [ITU93].)

This should make it clear that the call to the procedure and the nexstate is equivalent to the expanded connection, and therefore is correctness-preserving.

Repeated application

Some optimizing transformations can provide better results if they are applied multiple times. For example, some compression algorithms can be run on already compressed data to compress it even more.

In principle. my program will not improve the results with repeated application. All problems are resolved once and for all, and a second run should not find any problems to deal with.

The sole exception is if a connection that is transformed into a procedure contains connections itself. Normally, nested connections is not a problem, but if the connection is turned into a procedure **before** connections inside it are expanded, these connections will be put in a separate file, and not treated by the program. In the context of the procedure, these connections will be undefined.

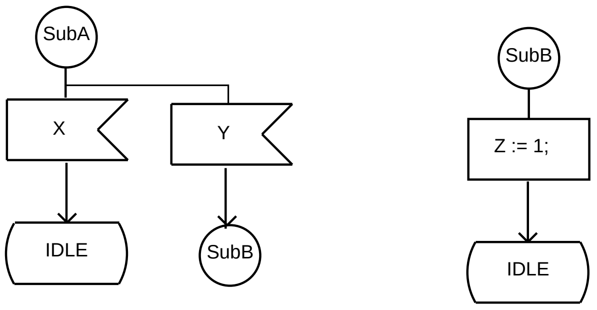


Figure 5.17 An example of nested connections

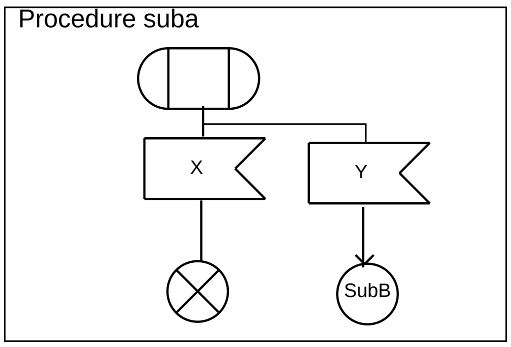


Figure 5.18 A generated procedure with an undefined connection

To avoid this, it might be necessary to run the program twice. First, treat all connections except the outer connection. In the second run, turn this connection into a procedure.

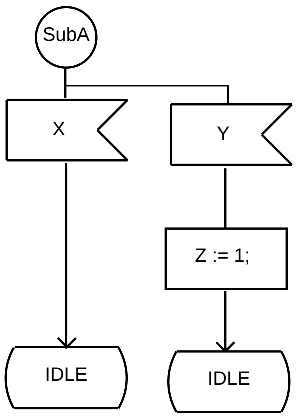


Figure 5.19 The connection, with the inner connection expanded

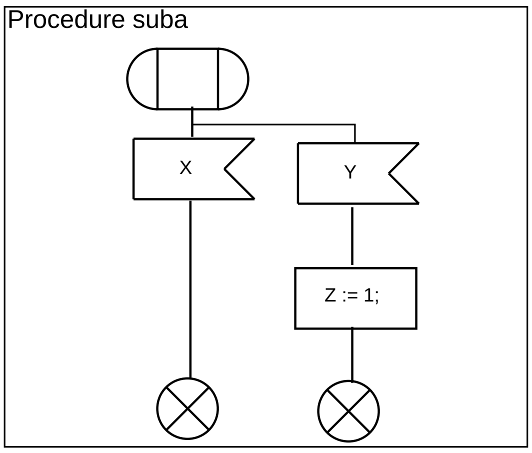


Figure 5.20 The procedure, correctly generated

CHAPTER 6

THE TEST CASE

Background

Since my program worked on the toy example, I knew that the approach worked in theory, and at least under ideal conditions. But of course, that did not prove that it would be usable in the real world. To test my program on a real-life example, I got hold of some SDL diagrams from the T.30 project in Ericsson.

The T.30 project made a design for a fax machine from the facsimile standard from ITU, T.30 [ITU93]. One interesting aspect of the project was that it used a transformational approach to generate SDL descriptions from the flowcharts in the standard. These flowcharts were action-oriented rather than state-oriented. Normally, action-oriented flowcharts does not translate well into SDL. In addition, the flowcharts used references (which translates into SDL connections) extensively.

Another interesting fact about the T.30 standard is that it is old. The history of ITU (CCITT before 1993) fax standards goes all the way back to 1968, with the T.2 standard. In 1980, the T.30 standard was introduced to replace the T.2 and T.3. The current version of T.30 is from 1993, and is just a update to the 1980 version. This means that the standard has gone through several cycles of maintenance and redevelopment, and probably have degraded structure, like Lehman’s second law predicts (see “History” on page 13). It would be interesting to see if an automatic transformation could make the descriptions more readable and maintainable.

The document I chose to work on was the description of Process phase, the top-most description of the fax behaviour. The original document is enclosed as Appendix C. The result after using `transformer.pl` on the SDL is enclosed as Appendix D.

New problems and solutions

As was expected, the T.30 diagrams introduced problems that the program was not prepared to handle. Mostly, these problems could be solved by adding more error handling in the procedures, but in two cases, invisible joins and overflow, new procedures had to be added to the program. All this made the program grow by nearly 50%. (From 805 lines in the version that handled the toy example to 1176 lines in the final version).

Invisible joins

Invisible joins are joins that do not use the join symbol, just a line with an arrow to the connection point. This is definitely not aesthetically pleasing or easy to read, from my point of view. Before I started working on process phase, had not even considered invisible joins as a problem. Fortunately, invisible joins are not much more difficult to expand than regular joins.

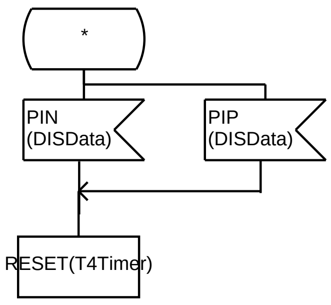


Figure 6.1 Example of invisible (graphical) join

The CIF code for the invisible join in Figure 6.1 will look like this:

```
input PIN
(DISData);
/* CIF Line (750,550),(750,650) */
/* CIF Label Invisible */
grst17:
.....
input PIP
(DISData);
/* CIF Line (1050,550),(1050,575),(750,575)*/
/* CIF Join Invisible */
join grst17;
```

Unfortunately, I think, the CIF standard also allows joining flowlines without using the join statement. The only clue for the program to find these joins is that one flowline ends without a proper statement, and that the line points to another line:

```
input PIN
(DISData);
/* CIF Line (750,550),(750,650) */
......
input PIP
(DISData);
/* CIF Line (1050,550),(1050,575),(750,575)*/
```

Then the join point must be found by looking at the coordinates of the joining line.

The removal of invisible joins could have been done inside the procedure that handled connections, but since the approach for invisible joins were a little bit different, I put it in a separate procedure.

The tactic was simple: on all “join invisible” statements, insert all the statements from the join point to the next nextstate or join. An example of a expanded join is shown in Figure 6.3.

Only expansion of joins with “proper” CIF statements was implemented. To expand the joins only specified by an arrow, the program would have to find the join point by the coordinates, a slightly more complex approach. Being a little short on time, I figured that the removal of well-specified joins would be enough to show the effect it would have on readability.

Overflow

In the toy example, the expansion of the connection created no overflow problems. But in process phase things were not so simple. Connections with branches became to wide and overwrote the flow to the right, or came beneath it and created trouble for the nextstate-alignment-algorithm.

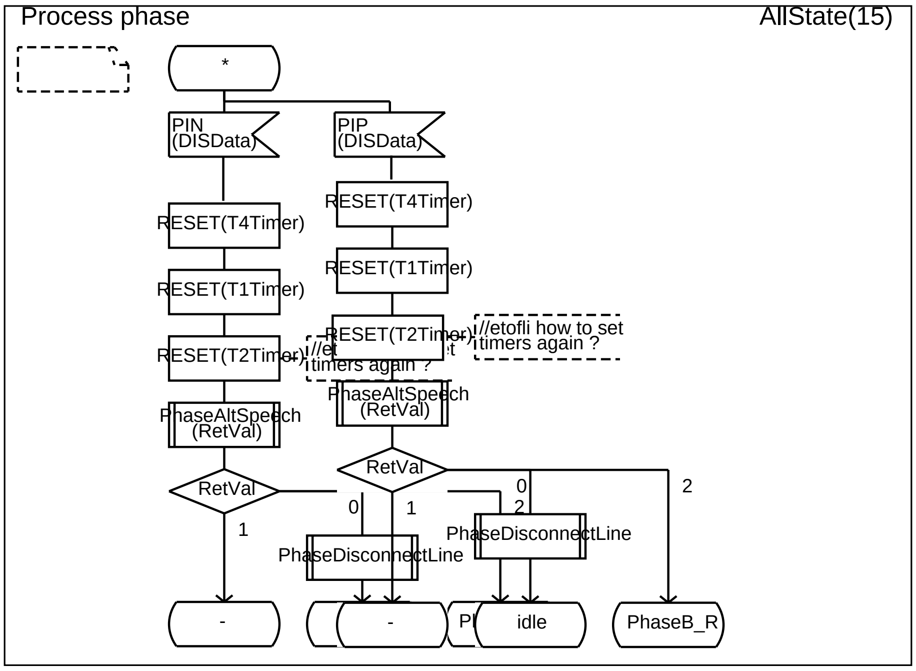


Figure 6.2 The result of expansion of an invisible join without overflow fixing

To remedy the overflow problem I had to create a new procedure, `resolvePageOverflow`, that checked all the pages after connection expansion. If the page contained too many flowlines, the excess flowlines were put on a new page. The procedure also shifted flows sideways.

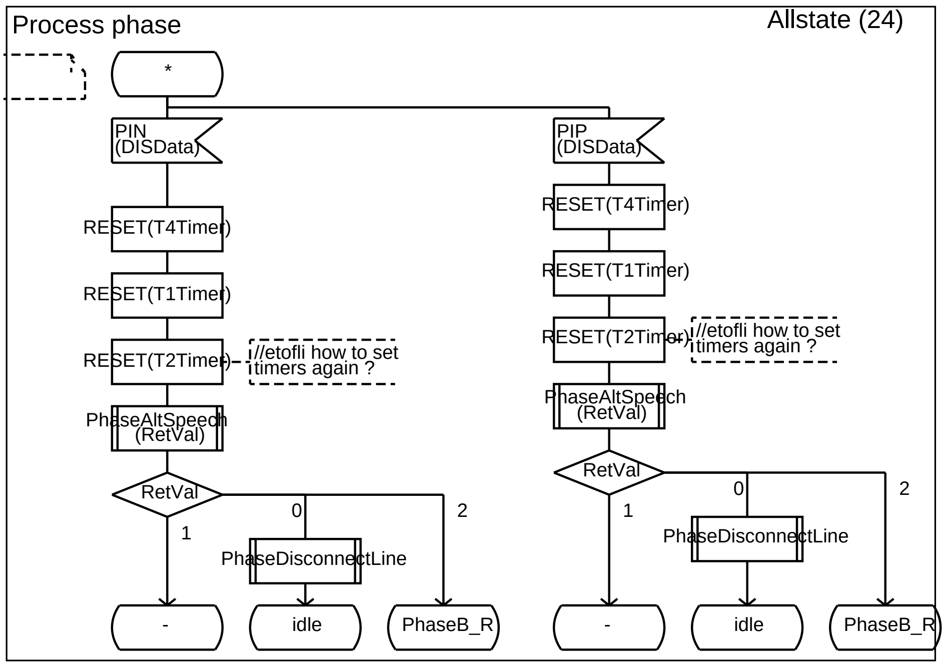


Figure 6.3 The state after expansion and applying `resolvePageOverflow`

Misplaced state symbols

For some reason, I had forgotten to check for misplaced state symbols, despite the fact that rule number one targeted exactly that. An example of a misplaced state symbol is shown in Figure 6.4. I did have a procedure to align newly created pages, but I had not thought of applying it to all pages.

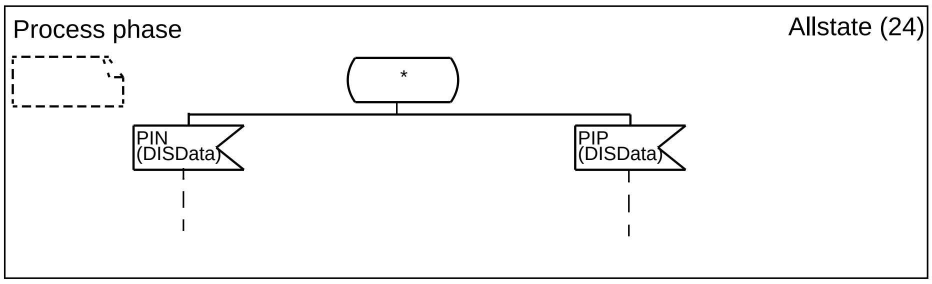


Figure 6.4 A misplaced state symbol

Instead of just aligning generated pages, the `alignPages` procedure was used on all pages. The procedure moved the state symbol to the upper left corner (coordinates 300,100). If necessary, the procedure also shuffled the rest of the symbols to make the first input symbol appear 250 pixels directly below the state symbol. An example of a correctly placed state symbol is shown in Figure 6.3.

Long variable and signal names

The identifier names were considerably longer in the T.30 test case than in the toy example, and often went past the symbol borders. If two flow-lines were too close together, the names would overwrite each other. It was obvious that the spacing between flowlines had to be increased, and that it should be possible to adjust to the diagrams in question.

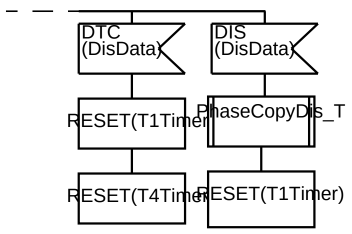


Figure 6.5 Overlapping signal names

Fortunately, this problem could be solved by the `resolvePageOverflow` procedure. I simply added a preference variable in the beginning of the program so that the spacing between flowlines could be adjusted to cater for the longest variables in the diagram.

Empty pages after expansion

If a page contained only a connection definition, the page would become empty if the connection was expanded (and the definition deleted). Since this did not occur in the toy example, I had not thought of this problem at all.

A small and simple procedure was added to delete pages that had become empty. The procedure checked if the pages had content or not, and if not, deleted the pagebreak and the page declaration.

Still unsolved problems

Because I only had a limited amount of time available for the programming, I had to stop the development of the program before all problems were solved. The problems I left unsolved were mostly those that would require a more advanced understanding of the SDL file.

Misplaced comments

In my toy example, the comments were conveniently placed where they could not cause any problems for the other symbols. In process phase, however, I encountered two placements of comments that caused problems:

1. Comments above the state symbol:
   When the state symbol was moved to the top of the page, these comments would disappear above the page.
2. Comments below join symbols:
   When the joins were expanded, the comments would overlap with the new symbols. An example of this can be seen on page 143.

The solution to this problem would have to be radically different than the other procedures in the program, as there is no simple transformation that can assure that a comment does not overwrite any symbols. The program would have to search for free space large enough to place a given symbol, and the program I made has no notion of "size" or "free space".

Overflow inside decisions

Unfortunately, the `ResolvePageOverflow` procedure only realigns whole flows, from the input symbol to the nextstate/join symbol(s). The procedure does not check for overflow inside decisions, which may occur if joins inside the decisions are expanded. An example of this is shown in Figure 6.6. This will also cause overlapping of long names to be left unattended if they appear inside decisions.

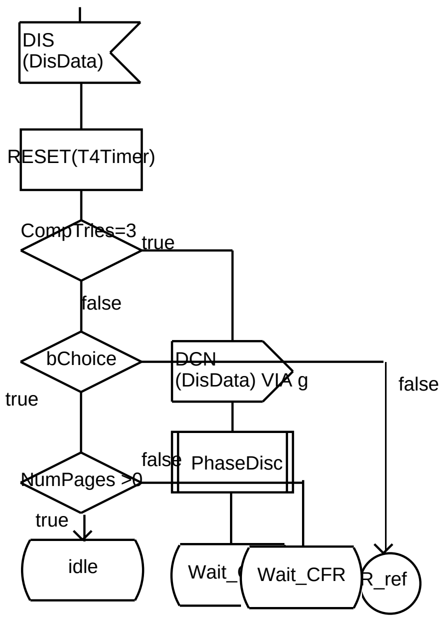


Figure 6.6 A (constructed) example of overflow within a decision

To solve this problem, the `resolvePageOverflow` procedure would have to be changed to also adjust the spacing between the flowlines inside decisions. This could be done within the current framework of the program by counting the number of nextstates/joins below decisions, and space the alternatives accordingly. For example, in Figure 6.6, the “true” branch of decision “Comptries=3” would be moved two “units” to the right, because there are three nextstates or joins below it.

A nicer approach would be to make the procedure recursive, so that the innermost decision was resolved first, and then the outer decisions. In that case, the “Numpages>3” decision in Figure 6.6 would be handled first, and its branches would take the two first spaces. The “false” branch of “bChoice” would then be moved to the right, to space number three.

Response from the T.30 team

The original and transformed SDL diagrams (effectively, appendix C and D) were sent to two members of the T.30 team. After a few weeks, I had a meeting with one of the developers. Our conversation revealed that he thought the program had really improved the readability of the diagrams.

He was most pleased with the alignment of nextstates and increased space between the flowlines. He also thought the removal of connections helped readability, especially when the connections were made into procedures.

He was sceptical of the expansion of connections in the diagrams, for two reasons. Firstly, the names of the connections disappeared. The names were important, because they corresponded to names in the T.30 standard. This was a typical case of restructuring destroying meaningful information, as described by Calliss (see “Problems with restructuring / Program transformation” on page 32). The developer proposed that the program should create comments with the name of the connection at the place of expansion.

Secondly, the expansion of connections, both regular and invisible, created much duplicate code. The developer would rather have a function that detected duplicate code and created procedures, to reduce redundant code.

All in all, the developer was quite satisfied with the results, but would like to see some improvements before using the program in his work.

CHAPTER 7

RESULTS & FURTHER WORK

Results

My literature studies, the development of the transformational program, and the use of it on the test case provided me with a lot of new experience and insight. In the first part of this chapter the concrete results will be presented. The results also revealed that there are still a lot of things that could be done to improve on my work. In the second part of this chapter, I will present some areas that could benefit from more work.

The dictionary

Work on the dictionary began before any of the other parts of this thesis, as a domain analysis of software maintenance. During the writing of the thesis, it was supplemented with terms used in the thesis. As of now, it comprises over 60 definitions, and constitutes a fairly comprehensive dictionary for reengineering and maintenance.

The literature study

My study of the reengineering literature resulted in the writing of Chapter 2, “An Introduction to Reengineering”. Others, particularly Robert Arnold [Arnold93], have written better introductions to reengineering, but together with the dictionary, I think this chapter can be a genuine help to someone new to the field.

Connecting transformation to pretty printing

Theoretically, there are no problems with using program transformation to improve a program's readability, instead of the usual goal: efficiency. It is just a case of making the transformational rules identify undersireable constructs in their left-sides, and providing an equivalent, but more readable solution in their right-sides. In fact, program transformation has potential to improve the readability and maintainability more deeply than traditional pretty printing, since transformation can change the design and organization of a program, not just its layout.

The rules

It turned out to be more difficult than expected to produce interesting rules that could be applied automatically.

This stems from the fundamental problem with automatic restructuring - to add meaning you need human input. This fact is reflected in the rules, all the correctable errors concern graphical details, and are close to trivial to correct. All the rules that would make the design of the SDL better are merely detectable or just not detectable. The most interesting rule in my program, the one about connections, needs a human to tell it what to do.

The program

There were no major design problems in making `transformer.pl`. The problem was the ever-increasing complexity and workload required to make the program able to read SDL written in all kinds of ways. To deal with less-than-perfect (or flat out wrong) SDL, the program needed a substantial amount of error handling code. To cater for the quirks of the test case, the program size increased by nearly 50%. This shows that the program was not (and probably still is not) very robust.

The complexity of my program would probably have been significantly less if a parsing approach had been used instead of string matching. A large part of the program logic is concerned with searching back and forth in the file to find the beginning and end of different constructs. If the SDL had been parsed into an abstract syntax tree or mapped to an object oriented model, this would have been avoided. Nevertheless, the feasibility of making a readability-enhancing program based on transformational principles should be clear to the reader.

The Test Case -Improvement in readability

Generally, the developer I talked with on the T.30 project was quite satisfied with the improved readability of the diagrams.

He was pleased with the alignment of nextstates and increased space between the flowlines, and also with the transformation of connections into procedures.

He was sceptical of the expansion of connections in the diagrams, for two reasons. Firstly, the names of the connections disappeared. Secondly, the expansion of connections, both regular and invisible, created much duplicate code.

Further work

As with all other human projects, time was a limiting factor on the work I did on this thesis. With more time, both the quantity and quality could have been improved. There are at least three subjects which could benefit from more work.

Empirical testing of SDL readability

A fundamental question to answer is “which SDL constructs are actually most easy to read?” The rules used in this thesis are either taken form Bræk and Haugen’s book [Bræk93] or made up by me, and none of those are based on empirical studies.

A good example of a study of readability was made by Soloway, Sonar and Erlich in 1983 [Soloway83]. They studied which looping constructs (in Pascal) was most easy to read. They made 280 Pascal programmers, some novices, some experienced, create a textual plan for a program and then code it with their favourite construct. Soloway et. al. found that the programmers that used Pascal constructs that closely matched their mental image made fewer errors. They also found that the default Pascal constructs did not match the “natural” thinking of the novices. Studies like this on SDL constructs would be very useful in determining how SDL layout should be.

Improve transformer.pl

As it is today, my program is too unfinished to be used for anything but experimentation. But if the program was improved, it could become a useful tool for developers. Things that needs to be done:

- `resolvePageOverflow` must work inside decisions
  This procedure is not able to adjust the spacing in the flow below a decision (see “Overflow inside decisions” on page 78). This aspect must be changed to make the program generally usable.
- Better sensing of overlap and free space
  Today, the program has no notion of “free space”, only of the relative placements of symbols. To be able to move comments that overlaps with other symbols, or insert new comments, this feature must be implemented (see “Misplaced comments” on page 77).
- Beautifying of decisions
  For the spacing algorithm to work correctly, decisions need to have their alternatives directly below or to the right, as described in “decision layout”, page 49. This rule has not been implemented in the program, but that must be done in order to make the program work satisfactory on most SDL.
- Creation of procedures for connections with multiple returns
  The program should absolutely be able to create procedures of connections with more than one nextstate (see page 63).
- Changing of cryptic names, by user input
  When the program finds a cryptic name, the user should be have the opportunity of supplying a better name. All occurrences of the name should then be replaced by the better one.

- Option to create comments where the connections are expanded
  If the user wants it, the program should insert a comment where it expands a connection. The comment could be supplied by the user, or it could be the name of the connection.

Spread my results to the industry

It is fairly obvious that my program can't make a large impact on how developers work with SDL. That is far more likely to happen if the commercial tool developers started using the rules from this paper in their programs. A good way to spread my results is to send this paper (or just maybe an email) to Telelogic, Verilog, and Cinderella. Telelogic already has a pretty good tool for restructuring SDL diagrams, and could improve it by implementing the rules presented in this thesis.

CHAPTER 8

CONCLUSION

Goal accomplishment

The objective of this thesis was to show that it is feasible to automatically restructure SDL to achieve better readability, and to support this activity with theory from program transformation. This has mainly been achieved.

I have pointed out that program transformation of syntax and semantics, as well as layout, can be used to improve readability.

I have presented a collection of rules for readable SDL. Unfortunately, only a minority of the rules can be enforced automatically. Still, the set of rules is large enough be used in an experiment.

To test the applicability of these rules, I created a prototype program. The program was built upon transformational principles, with the rules programmed as different procedures.

The program worked fine on a constructed example, but ran into some problems on an example from real life. Obviously, the program needs more work to be of any practical use.

Lessons learned

Through the work on this thesis, I have learned a lot, about the reengineering field, and about doing scientific work.

I got my views confirmed that meaningful information can only be added by humans, not machines. I did hope that my program would make the SDL descriptions much more readable, but most of the interesting rules were too dependent on subjective judgement to be enforced automatically.

I also got a strong indication of the limits of string matching for transformation. My program worked fine for smaller substitutions, and to some degree on replacing join symbols with the complete connection. But the program ran into trouble with larger, more complex structures, like nested decisions. Here. a parsing approach would have been much more appropriate.

In the later stages of my work, with time running out, I have regretted that I used so much time in getting to know the field and working on the TTM3 project at Ericsson. If I had started the work on `transformer.pl` earlier, the program would have been much more complete by now. But on a grander scale, I see that the experience from the literature study and the real world project may be more valuable to me later than the experience in writing perl scripts.

CHAPTER 9

REFERENCES

[Arnold89] Arnold, R. S.
Article: Software Restructuring
IEEE Proceedings, vol. 77, no. 4, pp. 607-617, 1989

[Arnold93] Arnold, R. S.
Book: Software Reengineering
IEEE Computer Society Press, 1993

[Arthur88] Arthur, L.J.
Book: Software Evolution: the Maintenance Challenge
John Wiley & Sons, Inc., 1988

[Belzer85] Belzer, R.
Article: A 15 year perspective on Automatic Programming
IEEE Transactions on Software Engineering, vol. 11, no. 11, 1985

[Biggerstaff89] Biggerstaff, T.J.
Article: Design Recovery for Maintenance and Reuse
IEEE Computer, vol. 22, no. 7, pp. 36-49, July 1989

[Boyle84] Boyle, J, M. & Muralidharan, M. N.
Article: Program Reusability through Program Transformation
IEEE Transactions on Software Engineering, vol. 10, no. 5, 1984

[Brodie95] Brodie, M. L. & Stonebraker, M.
Book: Migrating Legacy Systems: Gateways, Interfaces and
the Incremental Approach
Morgan-Kaufmann Publishers, 1995

[Bræk93] Bræk, R. & Haugen, H.
Book: Engineering Real Time Systems - an object-oriented methodology using SDL
Prentice Hall International, 1993

[Böhm66] Böhm, C. & Jaccopini, G.
Article: Flow diagrams, Turing Machines and Languages with only two transformational Rules
CACM, vol. 9, no. 5, pp. 366-371, 1966

[Calliss88] Calliss, F.
Article: Problems with automatic restructurers
ACM SIGPLAN notices, vol. 23, no. 3, 1988

[Chikofsky90] Chikofsky, E. J. & Cross, J. H. III
Article: Reverse Engineering and Design Recovery: A Taxonomy
IEE Software, vol. 7, no. 1, pp. 13-17, 1990

[Chu93] Chu, W.
Article: A Reengineering Approach to Program Translation
Proceedings, IEEE Conference on Software Maintenance, pp. 42-50, 1993

[Corbi90] Corbi, T.A.
Article: Program Understanding: Challenge for the 90s
IBM Systems Journal, vol. 28, no. 2, pp. 294-306, 1989

[DOMIS96] Surer, S. L. et. al.
Online paper: Distributed Object Management Integration Systems (DOMIS) FY96 final report
URL: `http://www.mitre.org/research/domis/reports.html`
MITRE, Center For Integrated Intelligence Systems, 1996.

[Ellsberger97] Ellsberger, J. et. al.
Book: SDL. Formal Object-Oriented Language for
Communicating Systems
Prentice Hall Europe, 1997

[Hallstein89] Hallstein, S.O. et. al.
Article: Transformational Program Development -An Approach for
Translating SDL to CHILL
Proceedings, SDL 89 - The Language at Work
North-Holland, 1989

[IEEE83] Institute of Electrical and Electronics Engineers
Standards paper: ANSI/IEEE std 729-1983 -IEEE Standard Glossary of
Software Engineering Terminology
IEEE, 1983

[IEEE93] Institute of Electrical and Electronics Engineers
Standards paper: IEEE STD 1219-1993 -Standard for Software
Maintenance
IEEE, 1993

[FAMOOS98] FAMOOS (Espirit project 21975)
Online article: SCG / FAMOOS / Candidate Reengineering Patterns
URL: `http://www.iam.unibe.ch/~famoos/`
Framework-based Approach for Mastering Object Oriented Software
evolution, 1998

[Ganti95] Ganti, N. & Brayman, W.
Book: The Transition of Legacy Systems to an Distributed Architecture
John Wiley & Sons, 1995

[ISO92] International Organization for Standardization
Standards paper: Information Technology: software product evaluation:
quality characteristics and guidelines for their use
ISO, 1992

[ITU93] International Telecommunications Union
Standards paper: Recommendation Z.100
-CCITT Specification and description language (SDL)
ITU, 1993

[ITU96] International Telecommunications Union
Standards paper: Recommendation Z.106
- Common interchange format for SDL
ITU, 1996

[Jacobson91] Jacobson, I. & Lindström, F.
Article: Re-engineering of Old Systems to an Object Oriented Architecture
Proceedings, OOPSLA, pp. 340-350, 1991

[Kozaczynski92] Kozaczynski, W. et. al.
Article: Program Concept Recognition and Transformation
IEEE Transactions on Software Engineering, vol.18, no.12, pp.1065-1075, 1992

[Lano94] Lano, K. & Haughton, H.
Reverse Engineering and Software Maintenance
Mc Graw-Hill Book Company Europe, 1994

[Lehman80] Lehman, M.
Article: Programs, Life Cycles, and the Laws of Software Evolution
IEEE Proceedings, vol. 68 no.9 pp. 1060-1076, 1980

[Letovsky86] Letovsky, S. & Soloway, E.
Article: Delocalized PLans and Program Comprehension
IEEE Software vol. 3 no.3?, may 1986

[Miller87] Miller, J.C., & Strauss, R. M. III
Article: Implications of automatic restructuring of COBOL
ACM SIGPLAN notices, vol. 22, no. 6, 1987

[McCabe] McCabe, T.J.
Article: A complexity measure
IEEE Transactions on Software Engineering, vol.2, no.4, pp.308-320, 1976

[Olsen94] Olsen, A. et. al.
Book: Systems Engineering using SDL-92
North Holland, 1994

[Patterson94] Patterson, D.A. & Hennesy, J.L.
Book: Computer Organization & Design - the Hardware/Software Interface
Morgan Kaufmann Publishers, Inc., 1994

[Pfleeger98] Pfleeger, S.L.
Article: The Nature of System Change
IEEE Software, vol. 15 no. 3 pp. 87-90, 1998

[RENAISSANSE98] RENAISSANCE Consortium
Online paper: RENAISSANCE Framework
URL: `http://www.comp.lancs.ac.uk/projects/renaissance/project/Documents/Method_Framework/MethodFramework.html`
RENAISSANCE Consortium, 1997

[Rich90] Rich, C. & Wills, L.M.
Article: Recognizing a Program’s Design: A Graph-Parsing Approach
IEEE Software vol.7 no.1 pp. 82-89, 1990

[SINTEF98] Stiftelsen for Industriell og Teknologisk Forskning ved NTH
TIMe Electronic Textbook -Combined Dictionary for TIMe and COP.
SINTEF, 1998

[Sneed87] Sneed, H.M. & Jandrasics, G.
Article: Software Recycling
Proceedings. Conference. on Software Maintenance, pp.82-90, 1987

[Sneed95] Sneed, H. M.
Article: Planning the Reengineering of Legacy Systems
IEEE Software, vol. 12 no.1, 1995

[SRI95] Software Reuse Institute
Online article: SRI Reuse Glossary
URL: `http://sw-eng.falls-church.va.us/ReuseIC/pubs/`
SRI, 1995

[Stevens98] Stevens, P. & Pooley, R.
Online article: Systems reengineering patterns
URL: `http://www.fzi.de/ECOOP98/submissons/ecoop98-stevens-pooley.ps`
ECOOP Workshop on Experiences in Object Oriented Reengineering, 1998

[Ward95] Ward, M.
Article: A Definition of Abstraction
Journal of Software Maintenance: Research and Practice, vol. 7, no. 6, pp. 443-450, 1995

[Zuylen93] van Zuylen, H. J.
Book: The REDO compendium, Reverse Engineering for Software Maintenance
John Wiley & Sons, 1993

APPENDIX A

DICTIONARY

Abstract Syntax Tree (AST)
An abstract syntax tree is a tree in which each inner node represents an operator and each leaf node represents an operand. Superficial distinctions of form do not appear in syntax trees, and places ASTs on a higher level of abstraction than source code.

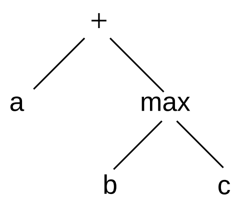


Figure A.1 The AST for `a + max(b,c)`

Abstraction
Informal: A general description, without details, of a problem, computer program, or system. Formal: A program S1 is an abstraction of another program S2 if each possible execution sequence for S1 consists of a subsequence of a possible execution for S2 [Ward95]. (Meaning that S1 has fewer states than S2, and all states in S1 is contained in S2.)

Adaptive maintenance
Maintenance activities undertaken to adapt a software system to a changed environment [IEEE83]. This includes maintenance as a consequence of changed hardware, software, business rules, government policies, or users requirements.

Application design
(Detailed) specification of the organization of the program(s) that constitutes the software of the computer system. This can include module partitioning, object design, and descriptions of algorithms.

Application generator
A type of tool that uses software designs and/or requirements to generate at least partial software applications automatically, such as program source code and program control statements [SRI95].

Architecture
An architecture is an abstraction of a concrete system representing:

- the overall structure of hardware identifying at least all physical nodes and interconnections needed to implement an abstract system.
- the overall structure of software identifying at least all software nodes, software communications and relations needed to implement an abstract system (in terms of processes, procedures and data) [SINTEF98].

Architectural design
The process og designing the architecture of a system.

BNF
The Backus-Naur Form (BNF) is a convenient means for writing down the grammar of a context-free language. It is compiled of a very simple structure that is similar to: "`<left term> ::= right-hand side`". This means that the left term side is defined by the right hand side. The symbol in the middle, ::=, is known as the meta-symbol. Terms enclosed in less-than and greater-than signs have definitions to explain them. These terms are called non-terminals.

CCITT
Consultative Committee on International Telephony and Telegraphy. In 1993, CCITT changed its name to ITU.

CIF
Common Interchange Format. A text-only format for describing layout of SDL diagrams. Specified by ITU in the Z.106 standard [ITU92].

CHILL
CCITT HIgh Level Language. A third generation language, most commonly used in the telecom industry. CHILL is specified by ITU in the Z.200 standard.

COBOL
A third generation language, most commonly used in administrative computing. COBOL was most popular in the 60s and 70s, but many of these systems are still in use. That makes COBOL systems interesting candidates for reengineering.

Compilation
The generation of assembler or machine code from source code.

Compiler
The program that performs compilation.

Computer program
A sequence of instructions suitable for processing by a computer [IEEE83].

Computer system
A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program [IEEE83].

Code generator
Synonym for source code generator [SRI95].

Conversion
Used as synonym for program transformation.

Corrective maintenance
Maintenance performed specifically to overcome existing faults [IEEE83]. Faults can have been introduced at any stage in the development of the software. Therefore “faults” includes design faults, logic faults and coding faults. Requirements faults usually have too large an impact on the system to be solved by corrective maintenance, often a complete reengineering of the system is necessary.

De-compilation
The generation of source code from machine code.

Design
The process of defining the software architecture, components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements [IEEE83].

Design recovery
The recreation of design abstractions from a combination of source code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains [Biggerstaff89].

Dynamic analysis
The process of evaluating a program based on execution of the program [IEEE83]. In a manual approach, one often read debug messages and try to trace the influence on output variables. Dynamic analysis tools often can discover memory leaks, processing bottlenecks and produce functional specifications.

Existing code
The source code of the running computer system. This code may or may not be old code.

Forward engineering, software development
The traditional process of moving from high-level abstractions and logical, implementation independent designs to the physical implementations of the system [Chikofsky90].

Functional requirements
Requirements that specify functions that a system or system component must be capable of performing [IEEE83].

Functional specification
A specification that defines the functions that a system or system component must perform [IEEE83].

Implementation
A machine executable form of a program, or a form of a program that can be translated automatically to machine readable form [IEEE83]. Both the source code and the machine code are considered part of the implementation.

Inverse engineering
The production of a functional specification from source code or machine code.

ITU
International Telecommunications Union. Specifies standards for international telecom, but also for some programming languages, such as SDL and CHILL.

Legacy system
Large (e.g. 10 million lines of source code), geriatric (e.g. more than 10 years old), and business critical systems [Brodie93]. A legacy system is usually very important to the organization, and very complex, or else it would have been replaced long ago. Legacy systems are often the target of software recycling and software salvaging.

Levels of abstraction
Documents at the same level of abstraction are roughly equally abstract relative to the system implementation. It’s normal to recognize four levels of abstraction: Requirements, design, functional specification, and implementation, in descending order of abstraction.

Machine code
The program, coded as instructions directly executable by a computer [IEEE83].

Maintenance personnel
The people responsible for keeping the system running, as well as developing it further. In short, they do software maintenance on the system.

Management
The people in charge of the organization using the computer system. Management makes guidelines for the users’ work process, and determines what the maintenance personnel should do on the system.

Module
A part of the system with defined boundaries [SINTEF98].

Non-functional properties
A non-functional property is a property which is not measurable in an abstract system. The non-functional properties can be related to the handling of abstract systems, for instance that they are flexible. More often they are related to the concrete system, and express physical properties such as size, weight and temperature. Performance, real-time responses and reliability are considered to be non-functional properties in TIMe, since they cannot be measured in the abstract systems [SINTEF98].

Non-functional requirements
Requirements and constraints on the system’s non-functional properties.

Old code
Existing code that cannot be easily understood, redesigned, modified or rewritten [Corbi89]. Old code isn’t necessarily old of age, just written in a complex and incomprehensible way.

Perfective maintenance
Maintenance performed to improve performance, maintainability, or other software attributes [IEEE83]. This includes adding extra functionality to the system and improving system performance.

Preventive maintenance is an important subgroup of perfective maintenance. Preventive maintenance is undertaken to make a system easier to maintain at a later stage. This includes all kinds of restructuring, both of code and design.

PR
Prose Representation. Textual representation of SDL. Often accompanied by CIF statements to describe a graphical SDL diagram, in a PR/CIF file.

Program slicing
The production of an extraction of the source code, with respect to the statements affecting a selected set of variables [Lano94]. This is done to ease the reading of the (interesting) source code, and to discover possible ripple effects.

Program transformation, program translation
The transformation of a program is viewed as a process of rewriting one program into another by repeated application of a set of transformation rules. A transformation rule has a left-side pattern, a right side pattern, and possibly some transformation conditions [Kozaczynski92].

Usually, source code is transformed into different, but semantically equivalent, source code. Transformation can be used in reverse engineering or forward engineering.

Program transliteration
To translate source code from one language to another, on a statement-by-statement basis.

Program understanding
The process of understanding the source code of a program, normally with the intention of changing it later. It's possible to gain understanding program both in a top-down and bottom-up manner.

Redevelopment engineering
A new development of a system's implementation from existing requirements. Optionally, the design is also remade.

Redocumentation
I will use this term to mean the generation of documentation from source code or machine code. This is a stricter definition than Chikofsky & Cross, who say that redocumentation can be done at all levels of abstraction, and have the following definition: "The creation or revision of a

semantic equivalent representation within the same relative level of abstraction” [Chikofsky90].

Reengineering
The examination and alteration of a subject system to reconstitute it in a new form and subsequent implementation of that form [Chikofsky90]. Reengineering involves reverse engineering, optionally restructuring or adaptation to new requirements, and a forward engineering phase. Some well-known synonyms for reengineering, often reflecting a specific reengineering purpose [Arnold93]:

- Software improvement
- Software renewal
- Software refurbishing
- Software modernization
- Software reclamation, software salvaging
- Software recycling
- Redevelopment engineering
- Reuse engineering

Some people spell reengineering with a hyphen: “re-engineering”. The fact that there is no uniform way of spelling this central word, is a telling example of the immaturity of the field. In this thesis, I will stick to Chikosfky & Cross’s spelling, without a hyphen.

(Re)modularization
Changing the module structure of a system. A new division of a system into modules often depend on analysis of systems component characteristics and coupling measures [Arnold93].

Requirements
Conditions or capabilities that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document [IEEE83]. Both functional and non-functional requirements exist.

Requirements specification
A specification that sets forth the requirements for a system or system component. Typically included are functional requirements, performance requirements, interface requirements, design requirements, and development standards [IEEE83].

Requirements recovery
The regeneration of a set of requirements for a system from the design and other sources. This might include studying the design documentation and interviewing experienced users and domain experts. Chikofsky & Cross calls this process “design recovery”, but I will use the term “requirements recovery”, to distinguish it from the recovery of system design.

Restructuring
Reorganization or changing of the system structure. The transformation from one representation to another at the same relative abstraction level, while preserving the subject system’s external behaviour [Chikofsky90]. According to Chikofsky & Cross, restructuring can be done at any level of abstraction. Redocumentation is considered restructuring at the implementation level, since it makes the source code or machine code more comprehensible.

Reuse engineering
Reengineering with the purpose of making a system more reusable at a later stage. Often, the system is (re)modularized or restructured to identify parts of the system that are good candidates for reuse. Reuse engineering is an example of preventive maintenance.

Ripple effect
The phenomenon that a change in one part of a program affect other sections in an unpredictable fashion, thereby leading to a distortion in the logic of the system [Takang96].

Reverse engineering
The process of analyzing a subject system to:

- Identify the system’s components and their interrelationships.

- Create representations of the system in another form, at a higher level of abstraction [Chikofsky90].

Different reverse engineering activities include de-compilation, inverse engineering, design recovery and requirements recovery.

SDL
Specification and Description Language. A programming language, used primarily by telecom, based on the ideas of extended finite state machines. SDL has statements and variables like all programming languages, but also has concepts of signals and processes, making it well suited for describing reactive, realtime systems. SDL was specified by ITU in the Z.100 standard [ITU93].

Software maintenance
Modification of a software product after delivery, to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [IEEE93]. Software maintenance can be divided into three categories: corrective maintenance, perfective maintenance and adaptive maintenance.

Software reclamation, software salvaging
Reengineering with the purpose of recovering lost information about a (legacy) system. The goal can be to find reusable objects, modules or subsystems (reuse engineering) or to recycle the whole design (software recycling).

Software recycling, software renewal
Reengineering with the purpose of recycling the design of old programs into a more modern form. According to Sneed & Jandrasics [Sneed87], the recycling process consists of the following steps:

1. Static analysis of existing code
2. Modularization
3. Internal restructuring of new modules in a design language
4. Manual optimization and adjustment of new modules. Optionally, adjust or extend the design to meet new requirements

5 Generation of new modular and structured code

Source code
The program, coded in a programming language that must be compiled, assembled, or interpreted before being executed by a computer [IEEE83].

Source code generator
A tool that uses software requirements and/or designs to automatically generate source code. An application generator generates entire applications, whereas a source code generator may generate smaller pieces of source code. Synonym for code generator [SRI95].

Static analysis
The process of evaluating a program without executing the program [IEEE83], for example analysis of the source code. Static analysis tools often produce call graphs, and find inactive code and uninitialized variables.

System erosion, system decay
The degradation of the system as it gets older. Usually, a system’s quality decreases, and its complexity increases after years of changes, fixes, and adaptations made during maintenance.

Transformational system
A program that runs transformations on program descriptions.

System
Unless otherwise noted, “system” will be used as a synonym for “compter system” in this thesis.

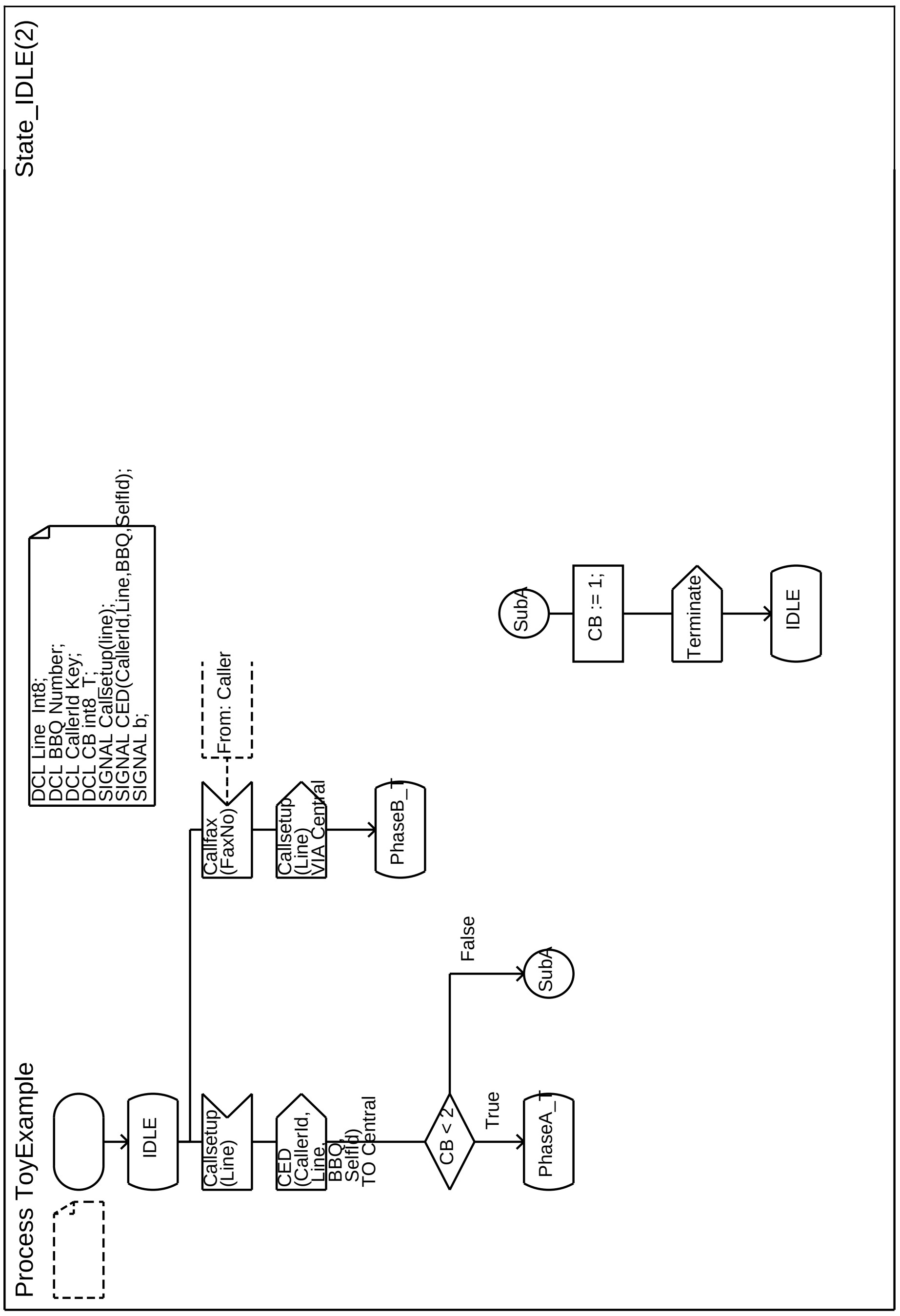
Users
The people that use the computer system as an aid in their work.

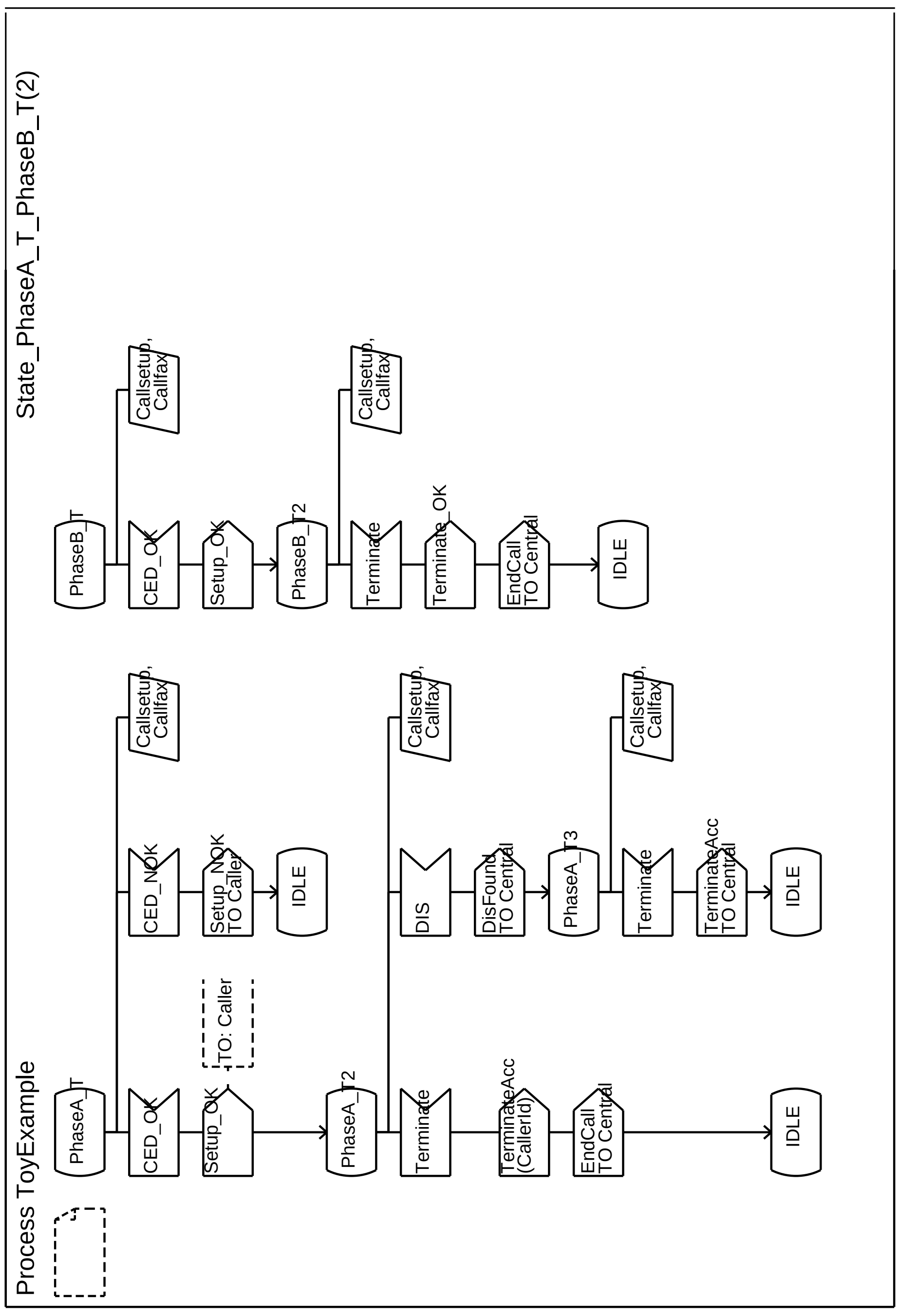
APPENDIX

B

SDL: Toy Example

This appendix contains the SDL diagrams from the toy example I used during the development of my program.

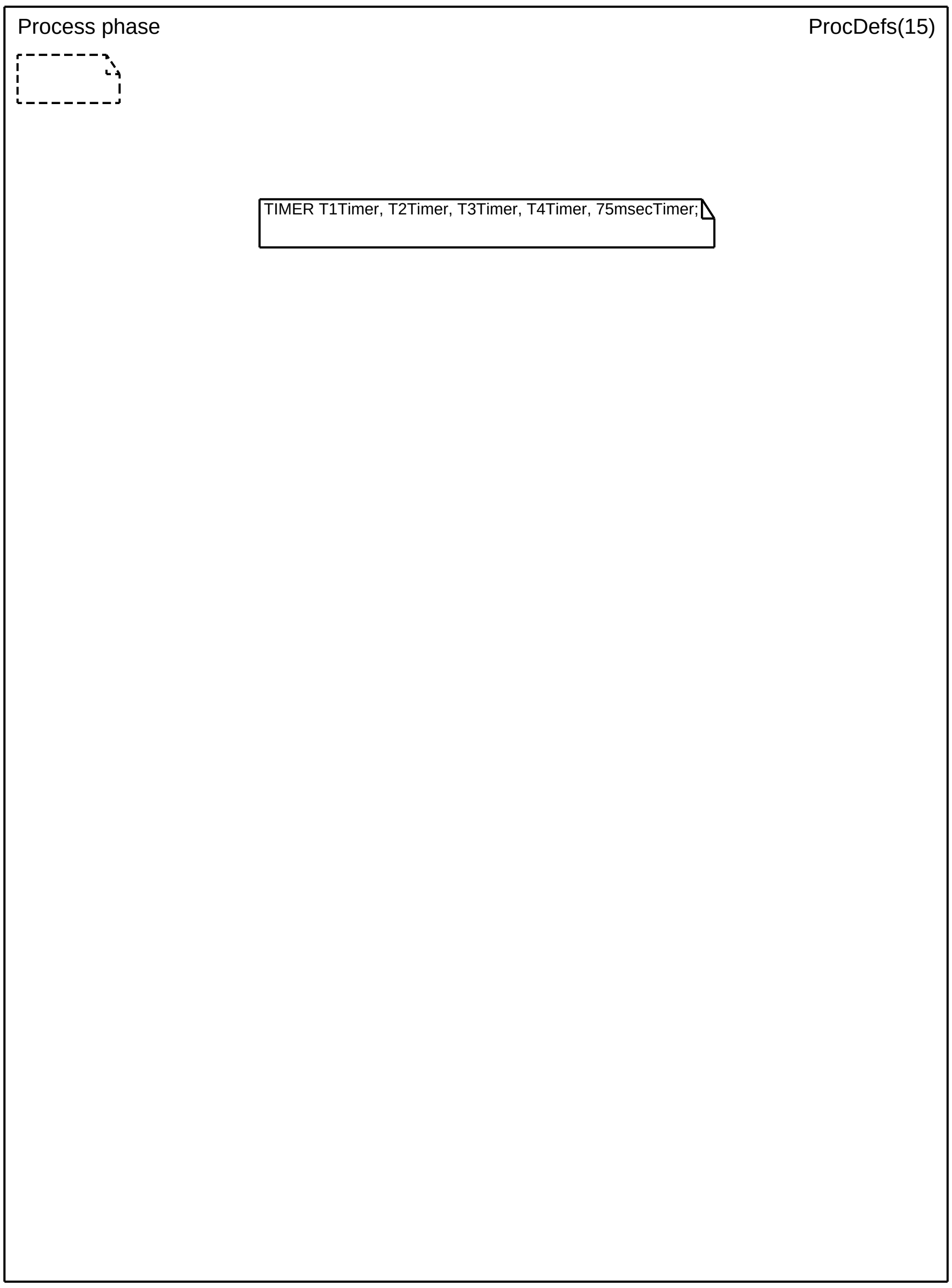

Process ToyExample
State_IDLE(2)
DCL Line Int8;
DCL BBQ Number;
DCL CallerId Key;
DCL CB int8 T;
SIGNAL Callsetup(line);
SIGNAL CED(CallerId,Line,BBQ,SelfId);
SIGNAL b;
IDLE
Callsetup (Line)
CED (CallerId, Line, BBQ, SelfId) TO Central
CB < 2
True
False
PhaseA_1
SubA
Callfax (FaxNo)
From: Caller
Callsetup (Line) VIA Central
PhaseB_1
SubA
CB := 1;
Terminate
IDLE


Process ToyExample
State_PhaseA_T_PhaseB_T(2)
PhaseA_T
CED_OK
Setup_OK
TO: Caller
PhaseA_T2
Terminate
TerminateAcc (CallerId)
EndCall TO Central
IDLE
CED_NOK
Setup_NOK TO Caller
IDLE
Callsetup, Callfax
DIS
DisFound TO Central
PhaseA_T3
Terminate
TerminateAcc TO Central
IDLE
Callsetup, Callfax
Callsetup, Callfax
PhaseB_T
CED_OK
Setup_OK
PhaseB_T2
Terminate
Terminate_OK
EndCall TO Central
IDLE
Callsetup, Callfax
Callsetup, Callfax

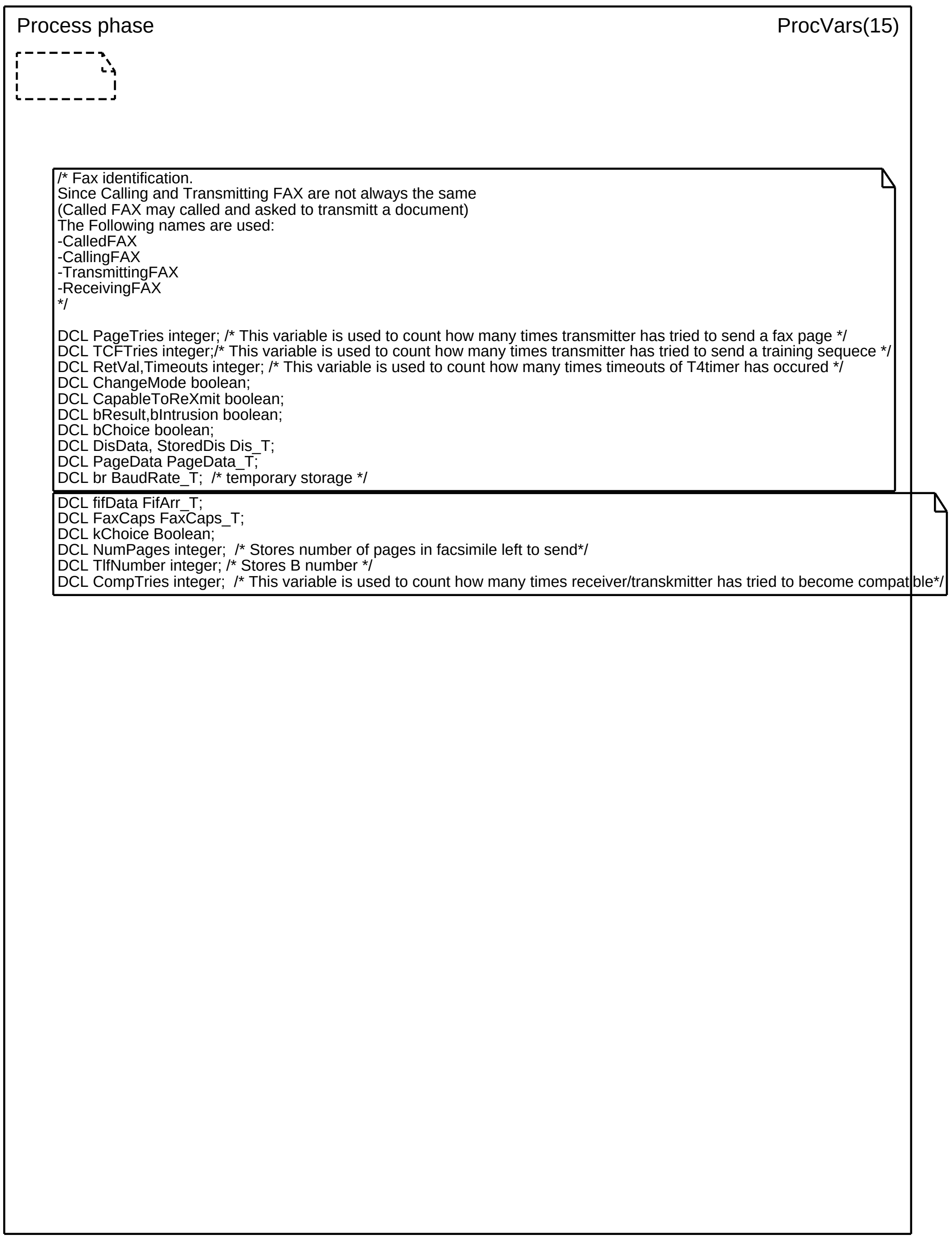
APPENDIX C

SDL: PROCESS PHASE (ORIGINAL)

This appendix contains the SDL diagrams used in the test case (Chapter 6), in their original form.

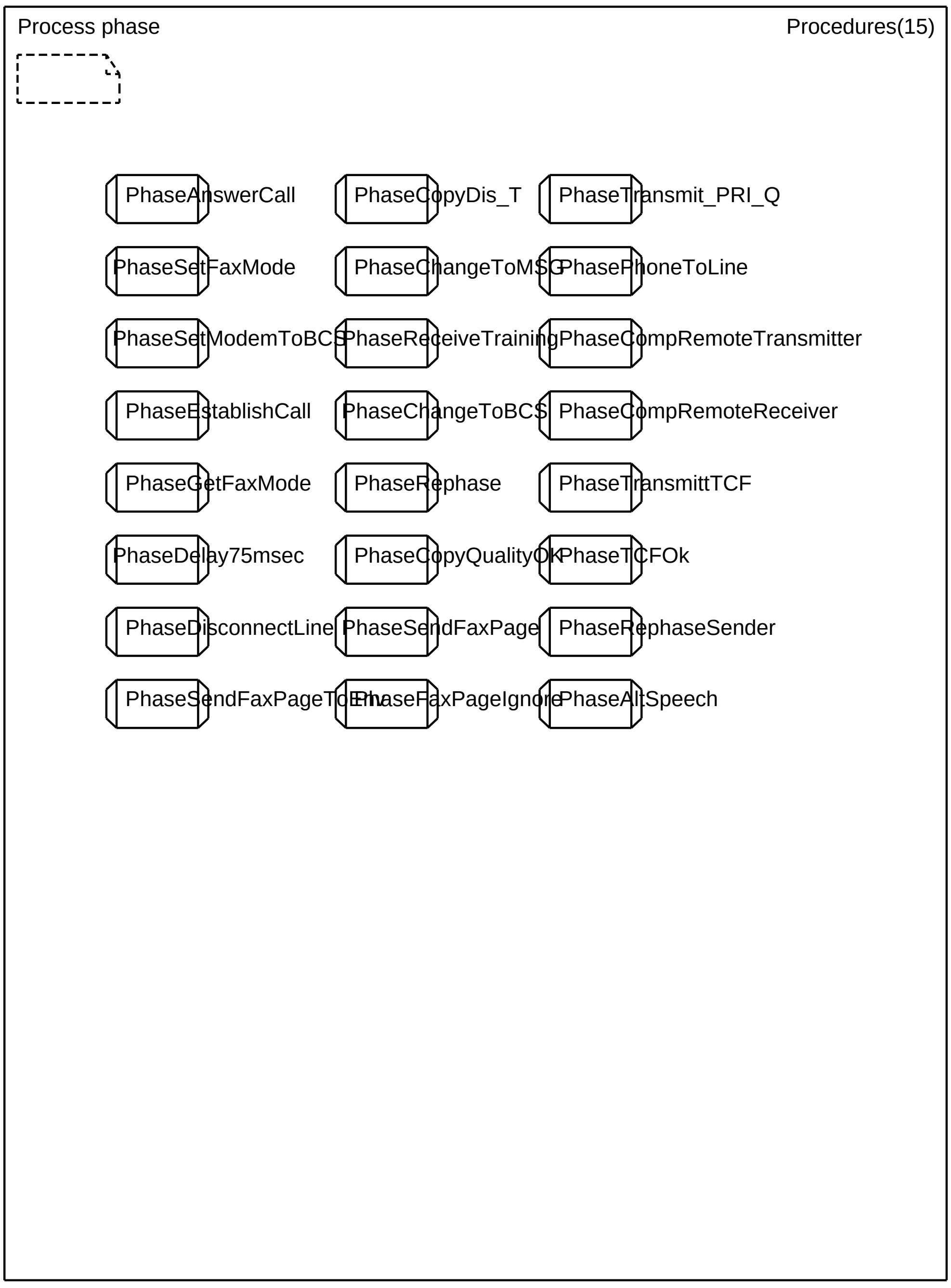


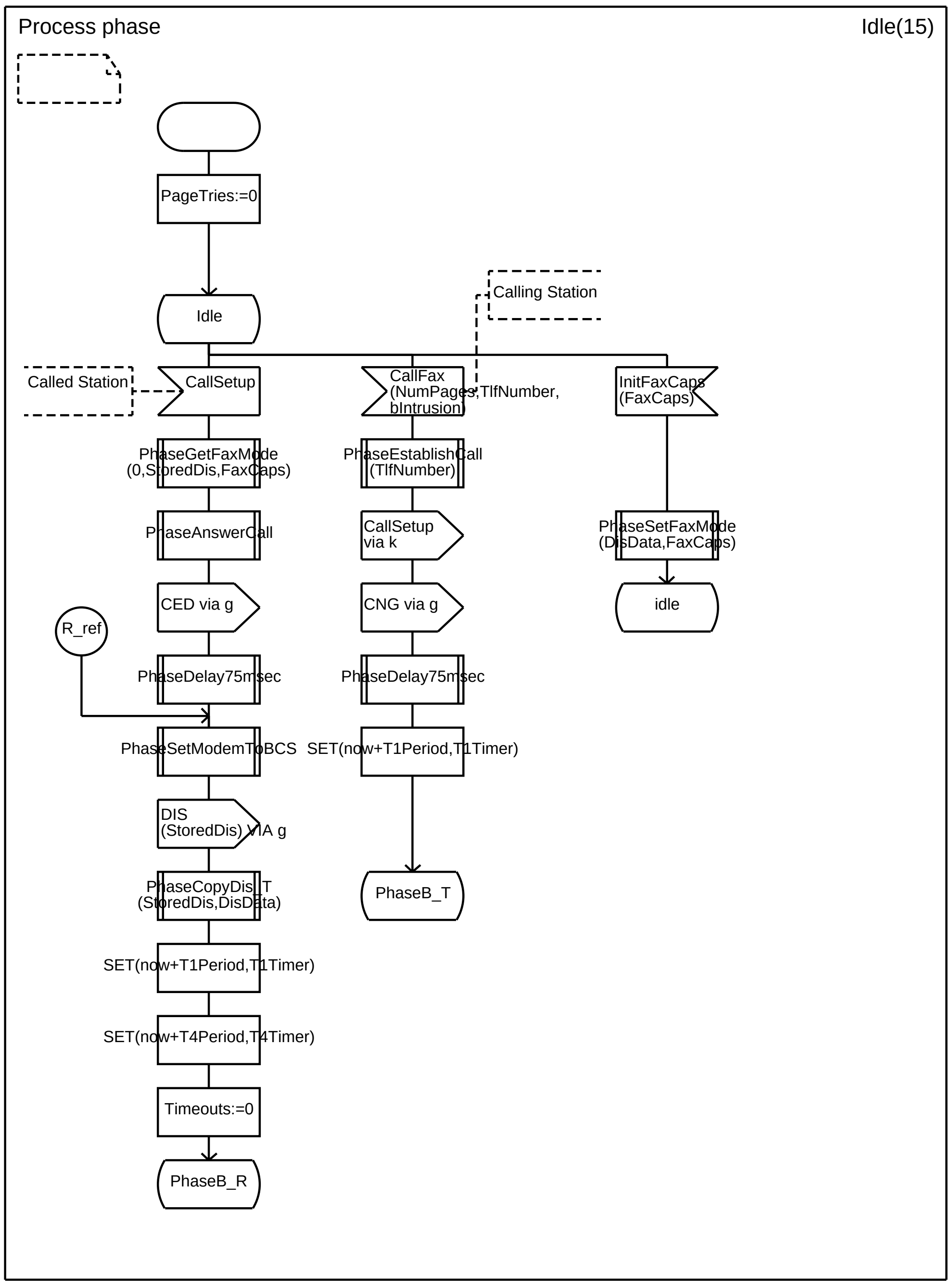
```
TIMER T1Timer, T2Timer, T3Timer, T4Timer, 75msecTimer;
```

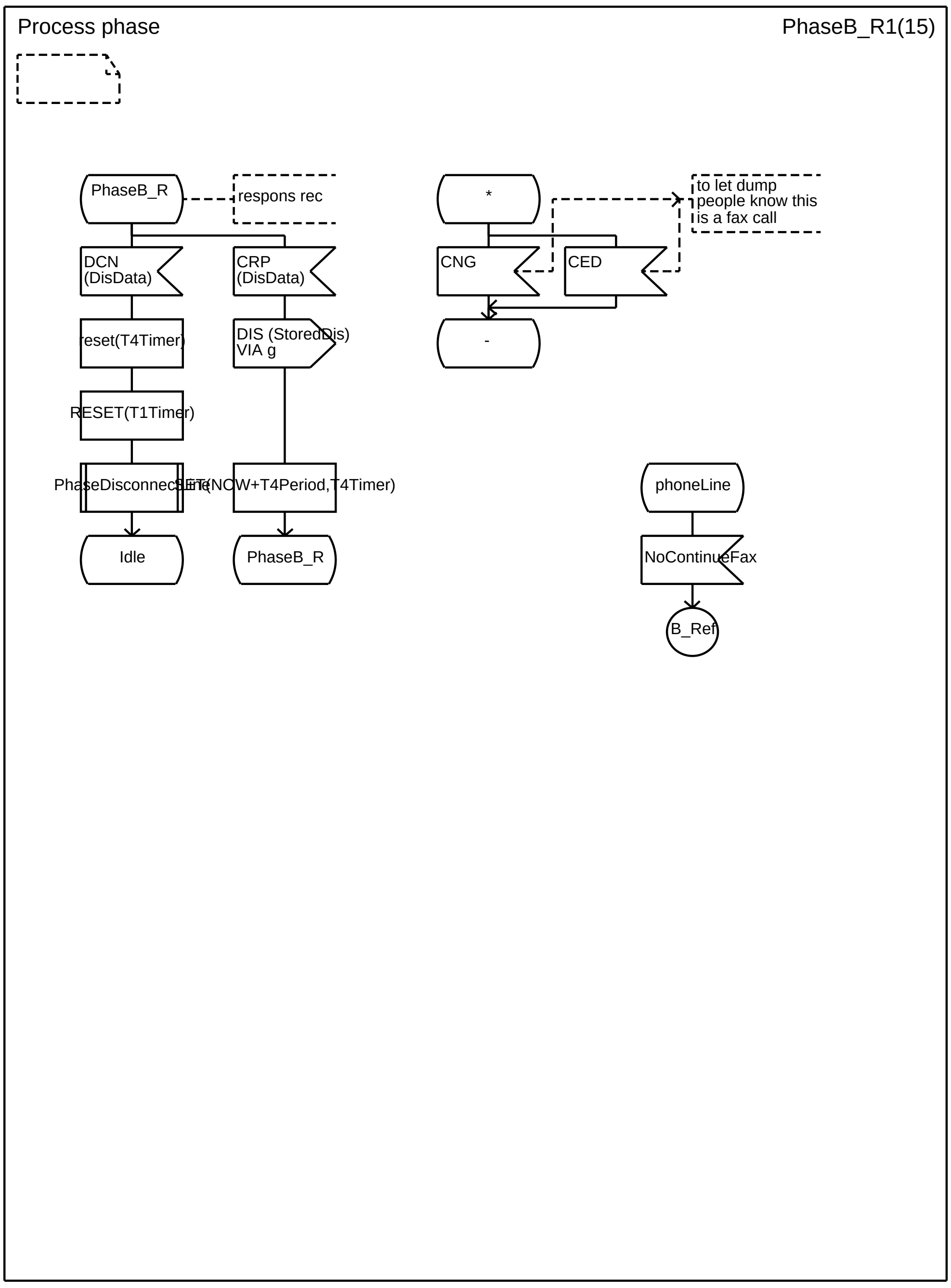


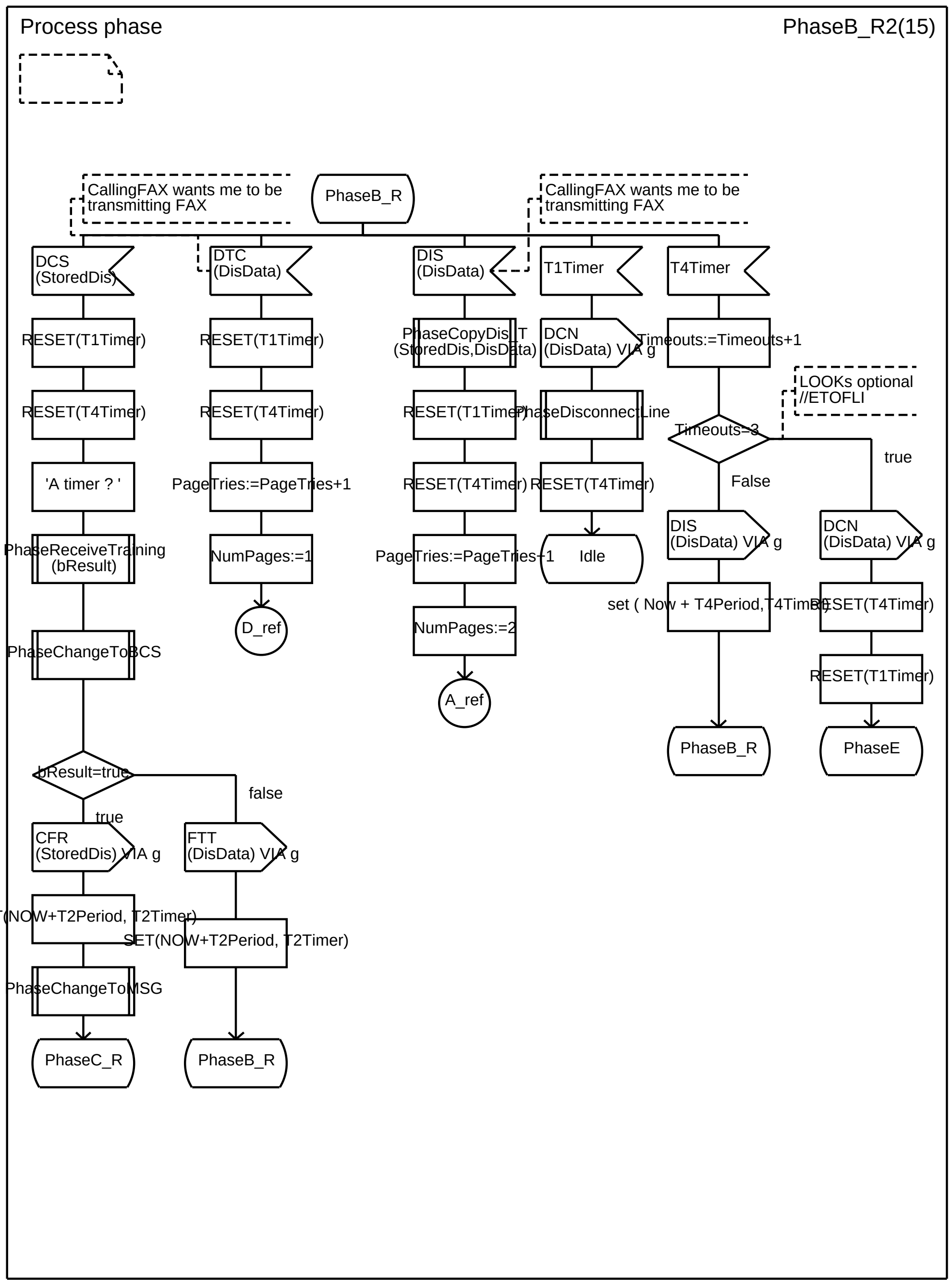
```
/* Fax identification.
Since Calling and Transmitting FAX are not always the same
(Called FAX may called and asked to transmitt a document)
The Following names are used:
-CalledFAX
-CallingFAX
-TransmittingFAX
-ReceivingFAX
*/

DCL PageTries integer; /* This variable is used to count how many times transmitter has tried to send a fax page */
DCL TCFTries integer;/* This variable is used to count how many times transmitter has tried to send a training sequece */
DCL RetVal,Timeouts integer; /* This variable is used to count how many times timeouts of T4timer has occured */
DCL ChangeMode boolean;
DCL CapableToReXmit boolean;
DCL bResult,bIntrusion boolean;
DCL bChoice boolean;
DCL DisData, StoredDis Dis_T;
DCL PageData PageData_T;
DCL br BaudRate_T;  /* temporary storage */
```

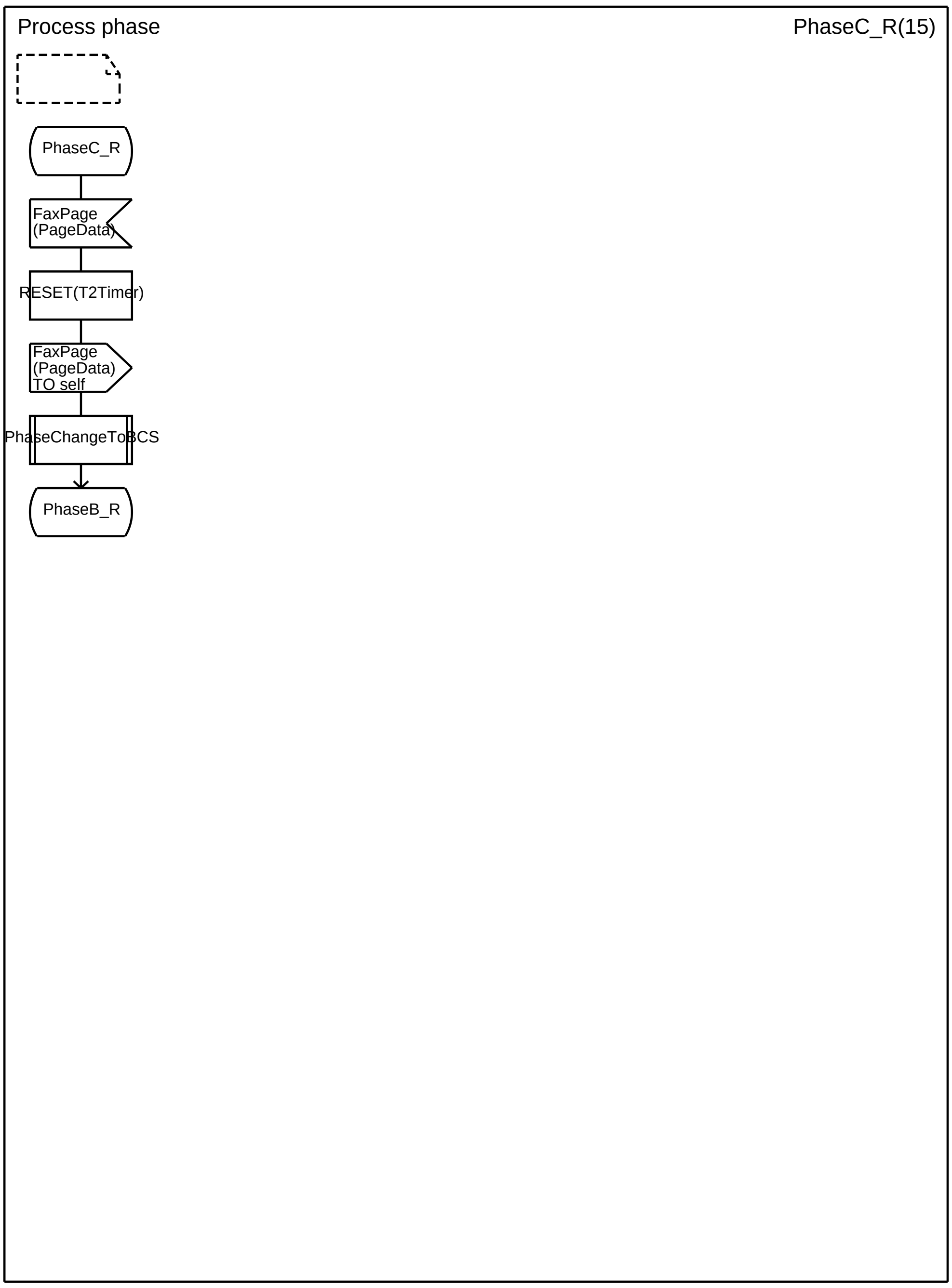
```
DCL fifData FifArr_T;
DCL FaxCaps FaxCaps_T;
DCL kChoice Boolean;
DCL NumPages integer;  /* Stores number of pages in facsimile left to send*/
DCL TlfNumber integer; /* Stores B number */
DCL CompTries integer;  /* This variable is used to count how many times receiver/transkmitter has tried to become compatible*/
```

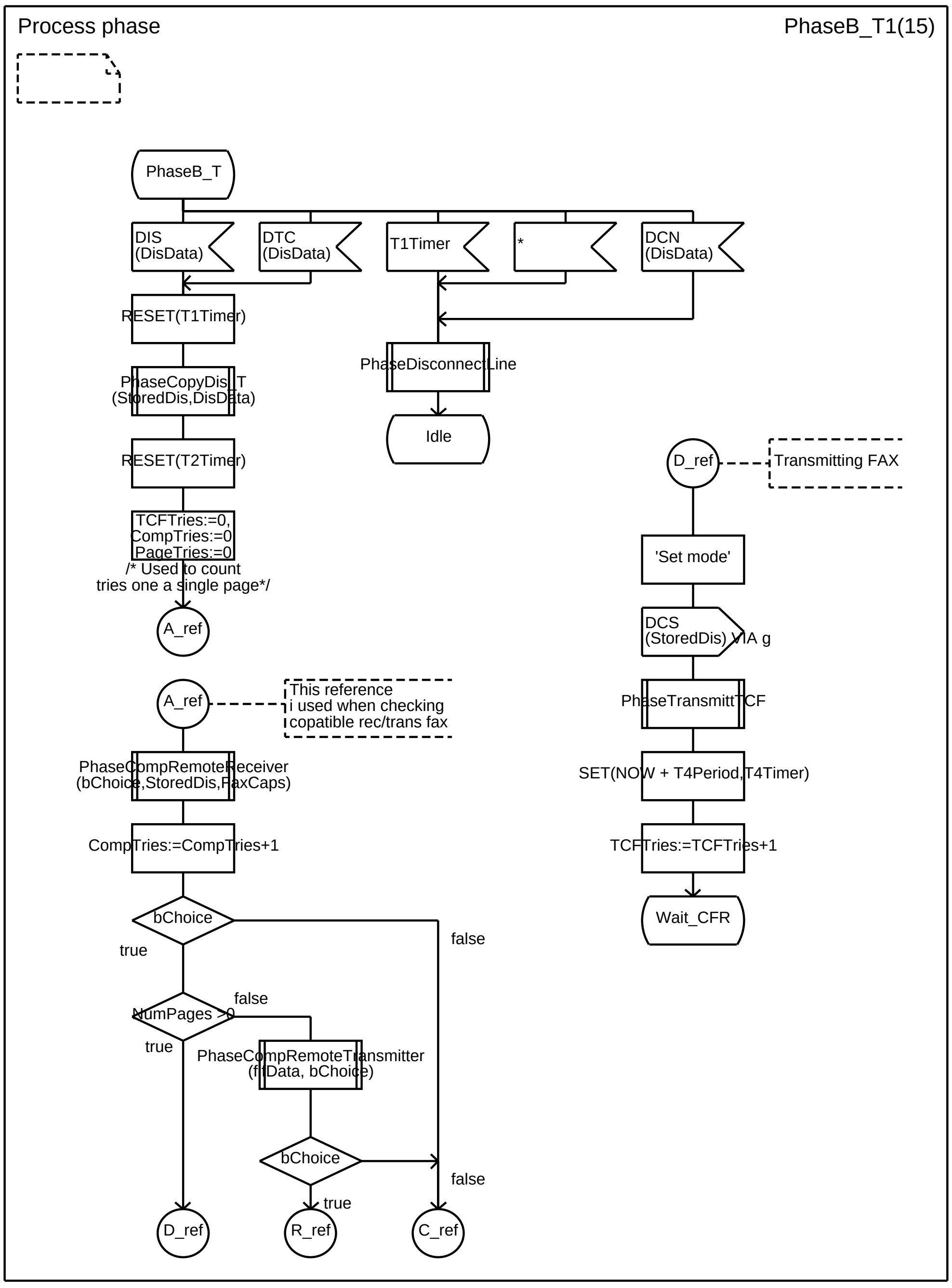

Process phase
Procedures(15)
PhaseAnswerCall
PhaseCopyDis_T
PhaseTransmit_PRI_Q
PhaseSetFaxMode
PhaseChangeToMSG
PhasePhoneToLine
PhaseSetModemToBCS
PhaseReceiveTraining
PhaseCompRemoteTransmitter
PhaseEstablishCall
PhaseChangeToBCS
PhaseCompRemoteReceiver
PhaseGetFaxMode
PhaseRephase
PhaseTransmittTCF
PhaseDelay75msec
PhaseCopyQualityOk
PhaseTCFOk
PhaseDisconnectLine
PhaseSendFaxPage
PhaseRephaseSender
PhaseSendFaxPageToEh
PhaseFaxPageIgnore
PhaseAltSpeech

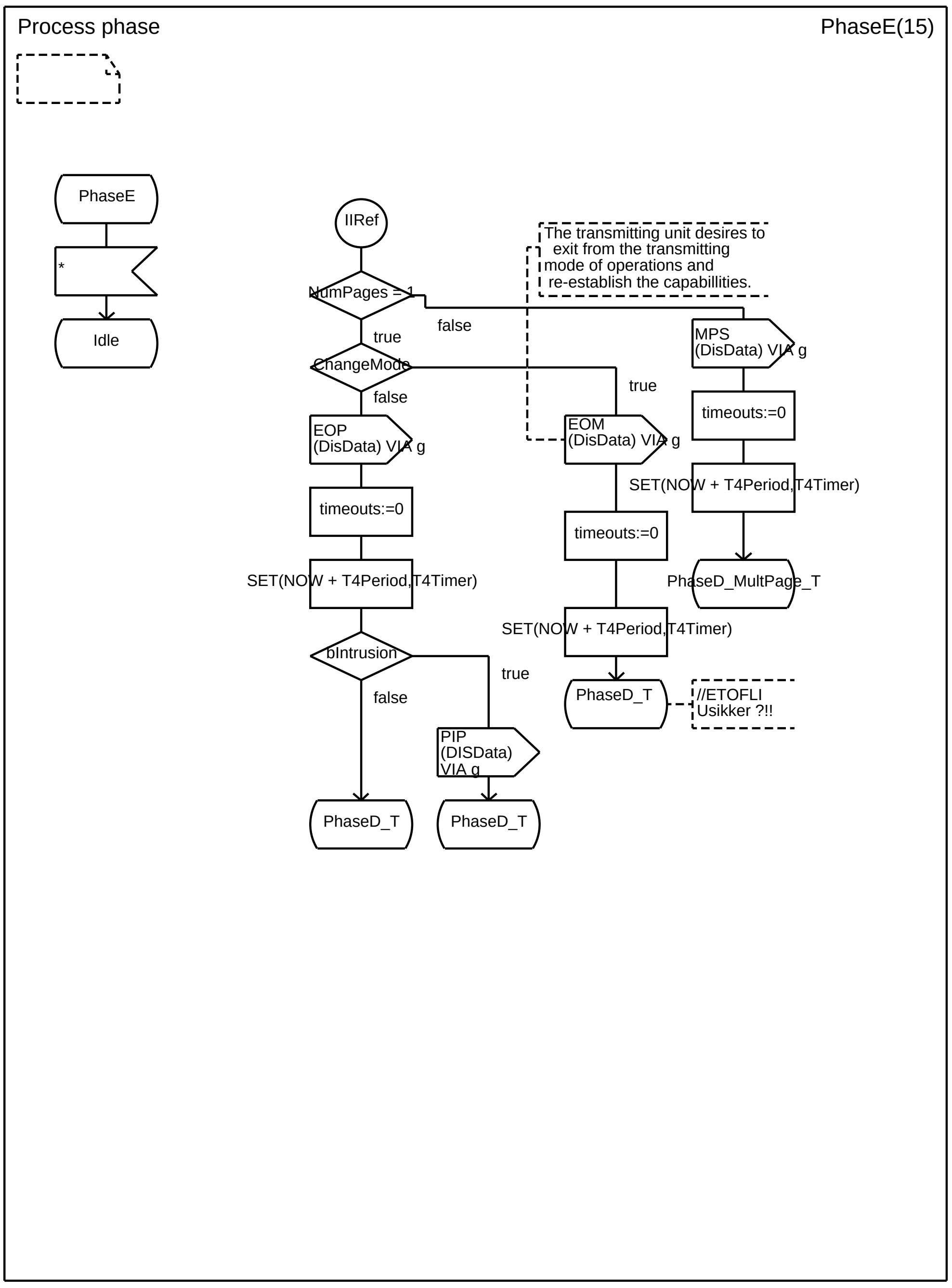

Process phase
Idle(15)
PageTries:=0
Idle
Calling Station
Called Station
CallSetup
CallFax (NumPages,TlfNumber, bIntrusion)
InitFaxCaps (FaxCaps)
PhaseGetFaxMode (0,StoredDis,FaxCaps)
PhaseEstablishCall (TlfNumber)
PhaseAnswerCall
CallSetup via k
PhaseSetFaxMode (DisData,FaxCaps)
CED via g
CNG via g
idle
R_ref
PhaseDelay75msec
PhaseDelay75msec
PhaseSetModemToBCS
SET(now+T1Period,T1Timer)
DIS (StoredDis) VIA g
PhaseCopyDis_T (StoredDis,DisData)
PhaseB_T
SET(now+T1Period,T1Timer)
SET(now+T4Period,T4Timer)
Timeouts:=0
PhaseB_R

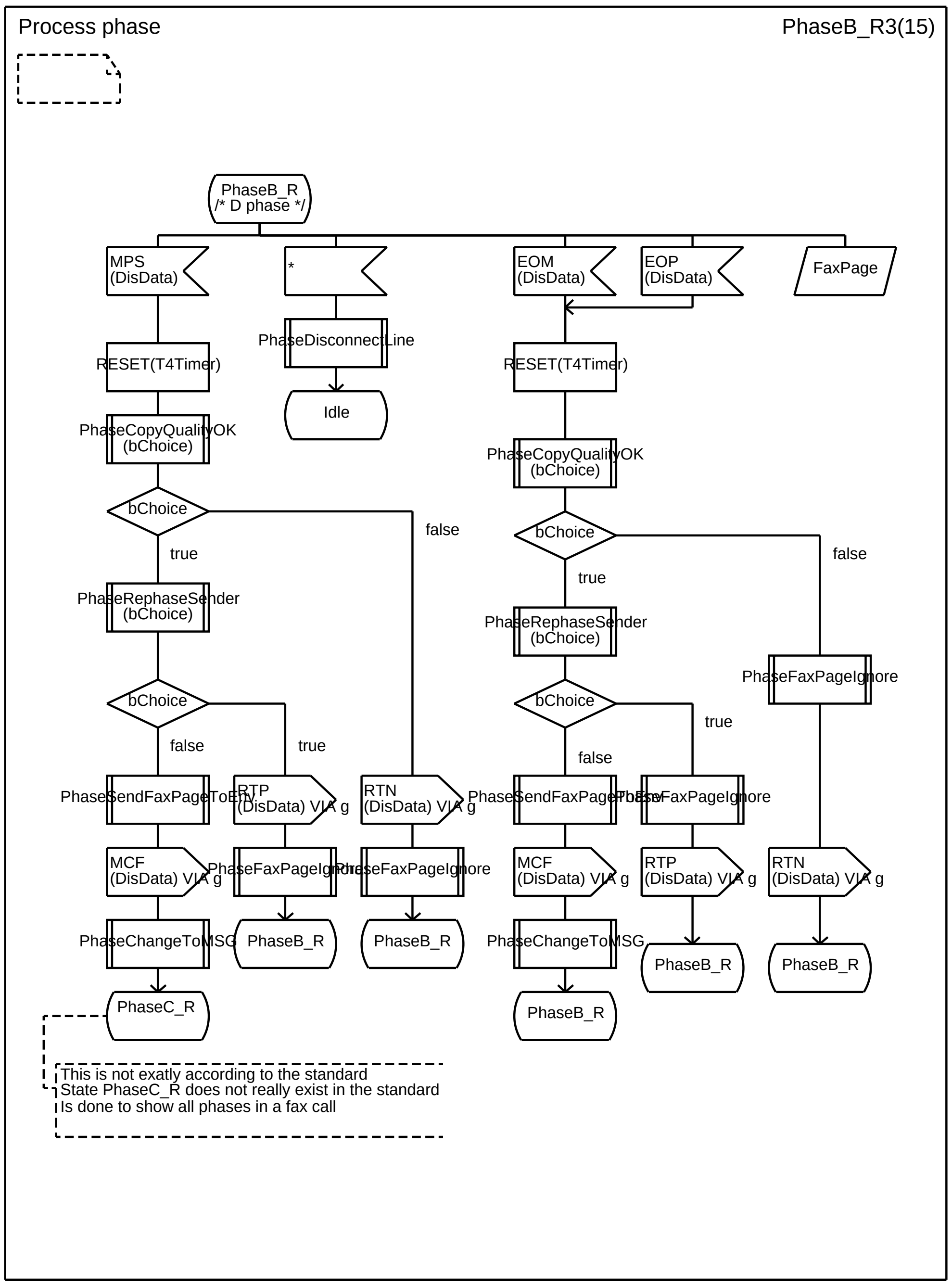

Process phase
PhaseB_R1(15)
PhaseB_R
respons rec
DCN (DisData)
CRP (DisData)
reset(T4Timer)
DIS (StoredDis) VIA g
RESET(T1Timer)
PhaseDisconnect
SET(NOW+T4Period,T4Timer)
Idle
PhaseB_R
*
to let dump people know this is a fax call
CNG
CED
-
phoneLine
NoContinueFax
B_Ref

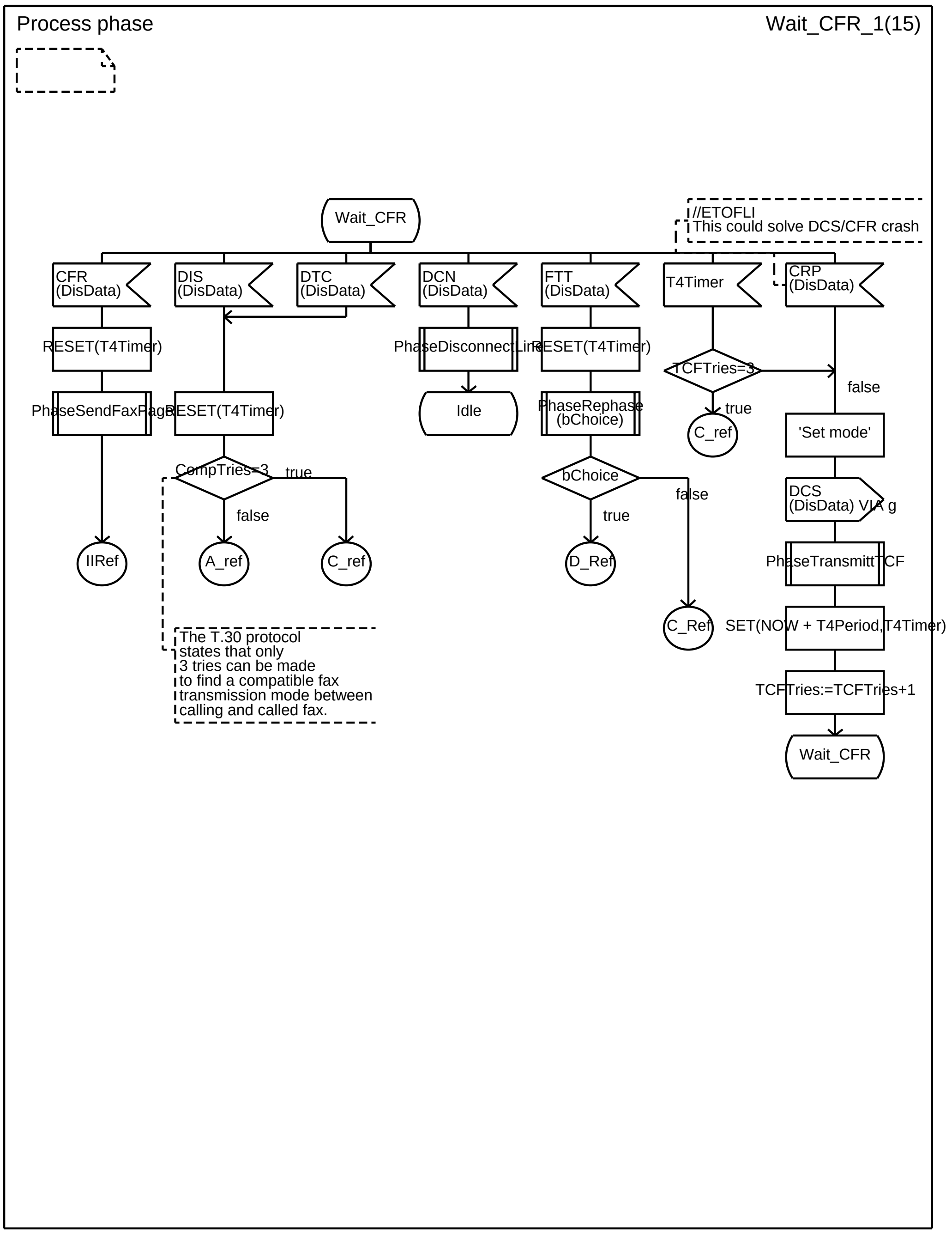

Process phase
PhaseB_R2(15)
PhaseB_R
CallingFAX wants me to be transmitting FAX
CallingFAX wants me to be transmitting FAX
DCS (StoredDis)
DTC (DisData)
DIS (DisData)
T1Timer
T4Timer
RESET(T1Timer)
RESET(T4Timer)
'A timer ? '
PhaseReceiveTraining (bResult)
PhaseChangeToBCS
bResult=true
true
false
CFR (StoredDis) VIA g
FTT (DisData) VIA g
SET(NOW+T2Period, T2Timer)
PhaseChangeToMSG
PhaseC_R
PhaseB_R
PageTries:=PageTries+1
NumPages:=1
D_ref
PhaseCopyDis_T (StoredDis,DisData)
NumPages:=2
A_ref
DCN (DisData) VIA g
PhaseDisconnectLine
Idle
Timeouts:=Timeouts+1
LOOKs optional //ETOFLI
Timeouts=3
False
DIS (DisData) VIA g
set (Now + T4Period,T4Timer)
DCN (DisData) VIA g
PhaseE

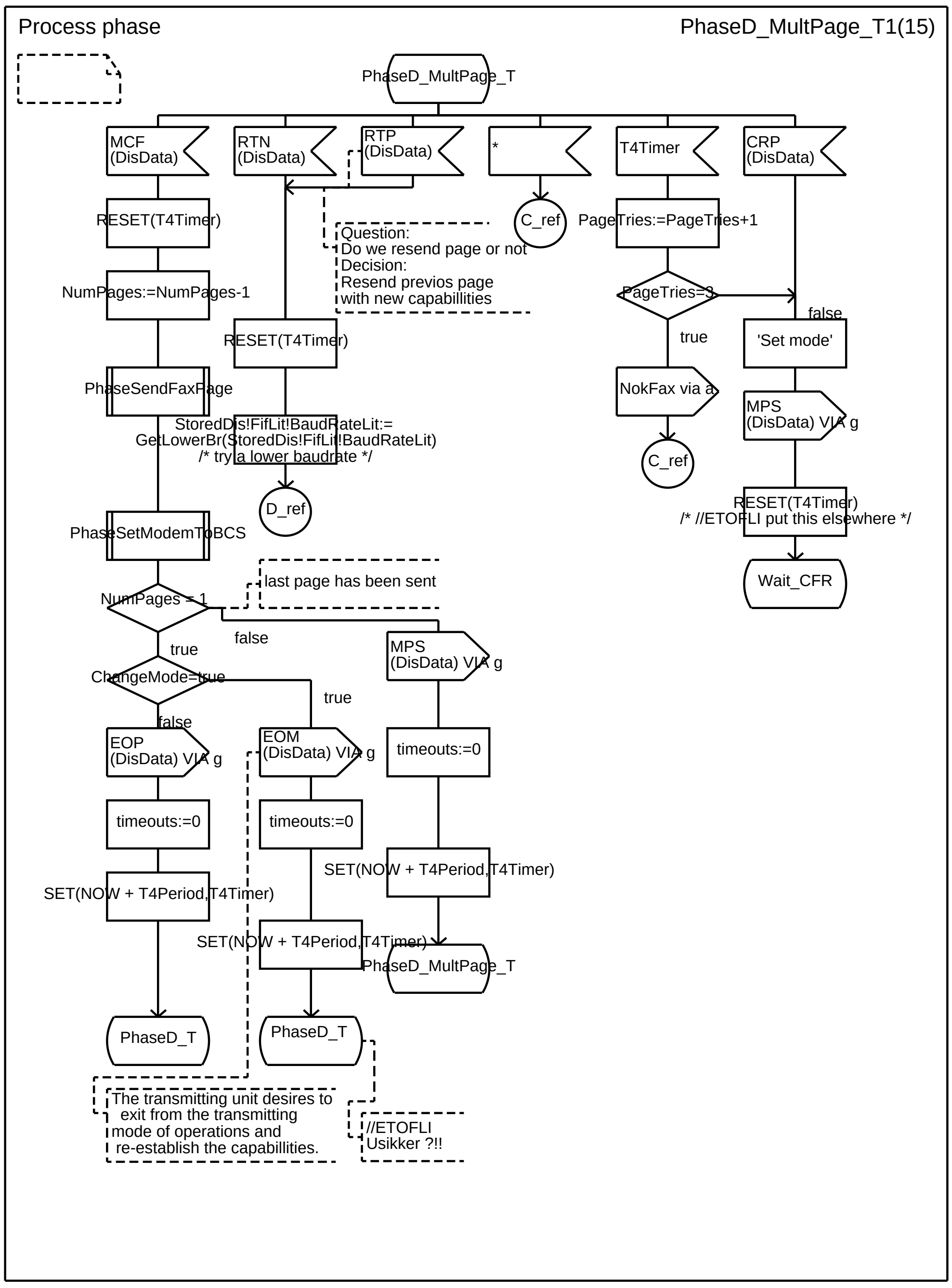

Process phase
PhaseC_R(15)
PhaseC_R
FaxPage (PageData)
RESET(T2Timer)
FaxPage (PageData) TO self
PhaseChangeToBCS
PhaseB_R

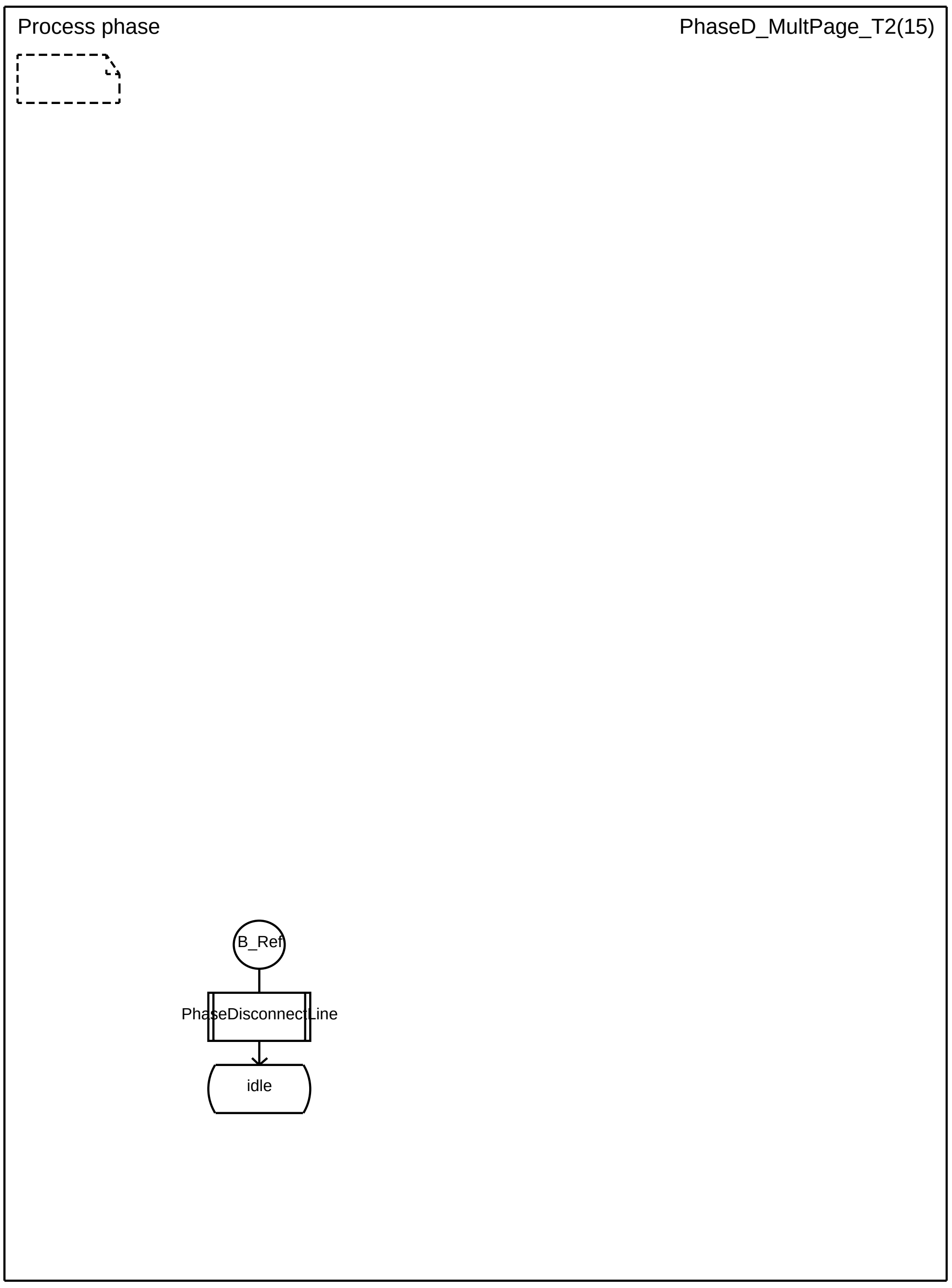

Process phase
PhaseB_T1(15)
PhaseB_T
DIS (DisData)
DTC (DisData)
T1Timer
*
DCN (DisData)
RESET(T1Timer)
PhaseCopyDis_T (StoredDis,DisData)
RESET(T2Timer)
TCFTries:=0, CompTries:=0 PageTries:=0
/* Used to count tries one a single page*/
A_ref
PhaseDisconnectLine
Idle
D_ref
Transmitting FAX
'Set mode'
DCS (StoredDis) VIA g
PhaseTransmittTCF
SET(NOW + T4Period,T4Timer)
TCFTries:=TCFTries+1
Wait_CFR
A_ref
This reference i used when checking copatible rec/trans fax
PhaseCompRemoteReceiver (bChoice,StoredDis,FaxCaps)
CompTries:=CompTries+1
bChoice
true
false
NumPages >0
false
true
PhaseCompRemoteTransmitter (fifData, bChoice)
bChoice
true
false
D_ref
R_ref
C_ref

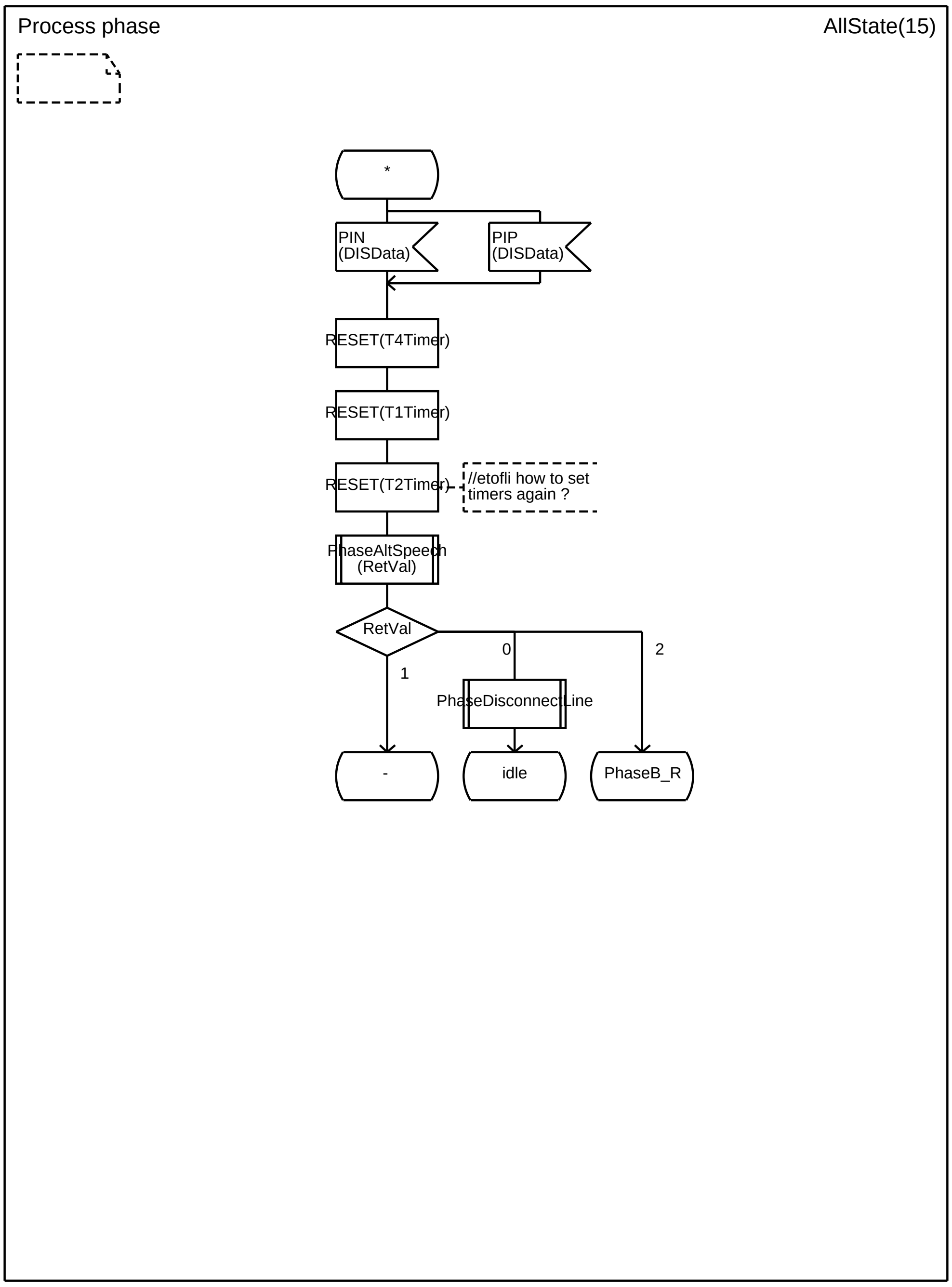

Process phase
PhaseE(15)
PhaseE
*
Idle
IIRef
NumPages = 1
false
true
ChangeMode
true
false
The transmitting unit desires to exit from the transmitting mode of operations and re-establish the capabillities.
MPS (DisData) VIA g
timeouts:=0
SET(NOW + T4Period,T4Timer)
PhaseD_MultPage_T
EOP (DisData) VIA g
timeouts:=0
SET(NOW + T4Period,T4Timer)
EOM (DisData) VIA g
timeouts:=0
SET(NOW + T4Period,T4Timer)
PhaseD_T
//ETOFLI Usikker ?!!
bIntrusion
true
false
PIP (DISData) VIA g
PhaseD_T
PhaseD_T

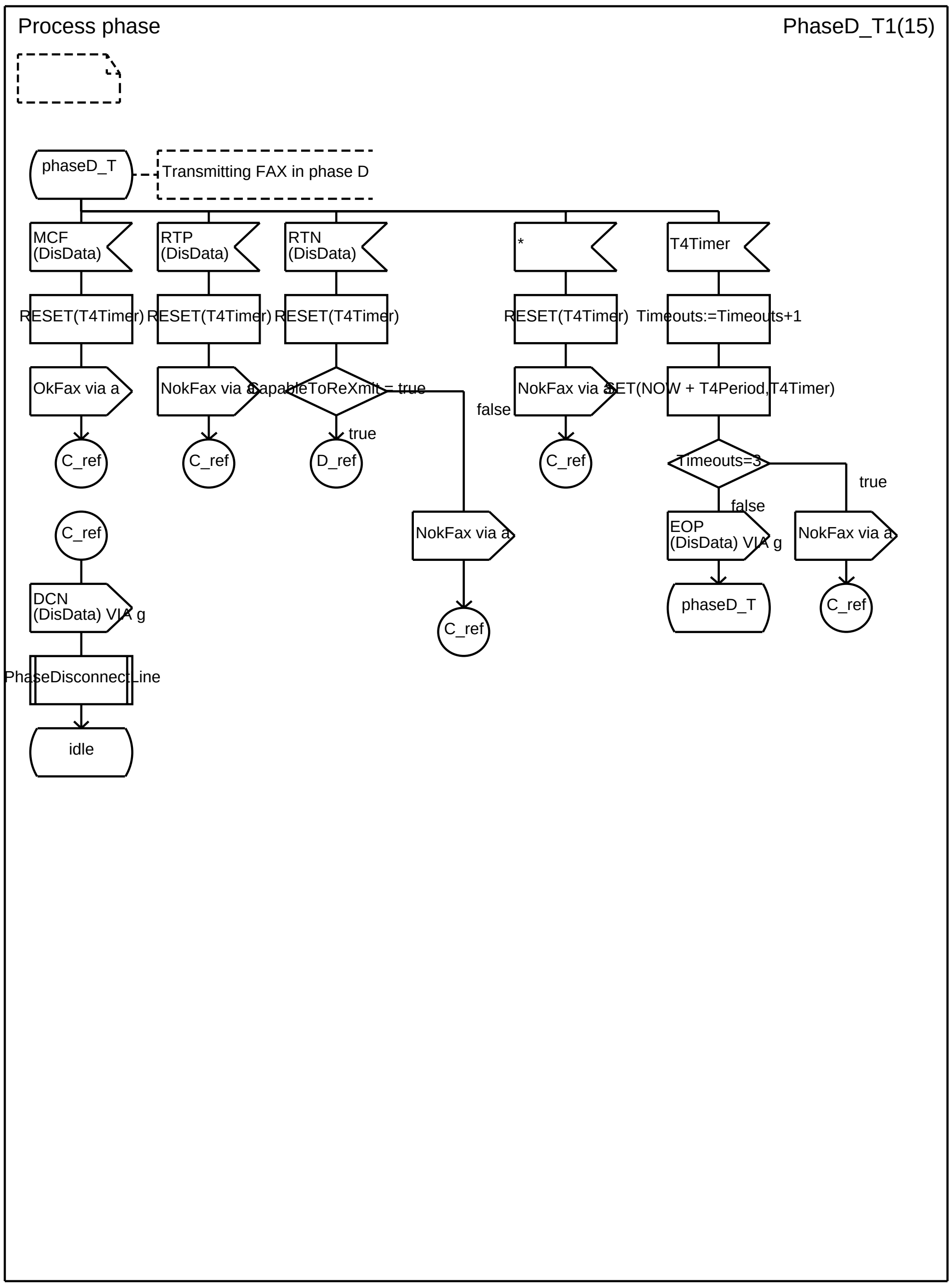

Process phase
PhaseB_R3(15)
PhaseB_R
/* D phase */
MPS (DisData)
*
EOM (DisData)
EOP (DisData)
FaxPage
RESET(T4Timer)
PhaseDisconnectLine
Idle
PhaseCopyQualityOK (bChoice)
bChoice
true
false
PhaseRephaseSender (bChoice)
PhaseSendFaxPageToEnv
RTP (DisData) VIA g
RTN (DisData) VIA g
MCF (DisData) VIA g
PhaseFaxPageIgnore
PhaseChangeToMSG
PhaseB_R
PhaseC_R
This is not exatly according to the standard
State PhaseC_R does not really exist in the standard
Is done to show all phases in a fax call


Process phase
Wait_CFR_1(15)
Wait_CFR
//ETOFLI
This could solve DCS/CFR crash
CFR (DisData)
DIS (DisData)
DTC (DisData)
DCN (DisData)
FTT (DisData)
T4Timer
CRP (DisData)
RESET(T4Timer)
PhaseSendFaxPage
IIRef
RESET(T4Timer)
CompTries=3
true
false
A_ref
C_ref
PhaseDisconnectLine
Idle
RESET(T4Timer)
PhaseRephase (bChoice)
bChoice
true
false
D_Ref
C_Ref
TCFTries=3
true
false
C_ref
'Set mode'
DCS (DisData) VIA g
PhaseTransmittTCF
SET(NOW + T4Period,T4Timer)
TCFTries:=TCFTries+1
Wait_CFR
The T.30 protocol states that only 3 tries can be made to find a compatible fax transmission mode between calling and called fax.


Process phase
PhaseD_MultPage_T1(15)
PhaseD_MultPage_T
MCF (DisData)
RTN (DisData)
RTP (DisData)
*
T4Timer
CRP (DisData)
RESET(T4Timer)
NumPages:=NumPages-1
PhaseSendFaxPage
PhaseSetModemToBCS
NumPages = 1
last page has been sent
true
false
ChangeMode=true
false
true
EOP (DisData) VIA g
EOM (DisData) VIA g
MPS (DisData) VIA g
timeouts:=0
SET(NOW + T4Period,T4Timer)
PhaseD_T
PhaseD_MultPage_T
The transmitting unit desires to exit from the transmitting mode of operations and re-establish the capabillities.
//ETOFLI Usikker ?!!
RESET(T4Timer)
StoredDis!FifLit!BaudRateLit:= GetLowerBr(StoredDis!FifLit!BaudRateLit) /* try a lower baudrate */
D_ref
Question: Do we resend page or not Decision: Resend previos page with new capabillities
C_ref
PageTries:=PageTries+1
PageTries=3
true
false
NokFax via a
'Set mode'
MPS (DisData) VIA g
RESET(T4Timer) /* //ETOFLI put this elsewhere */
Wait_CFR


Process phase
PhaseD_MultPage_T2(15)
B_Ref
PhaseDisconnectLine
idle


Process phase
AllState(15)
*
PIN (DISData)
PIP (DISData)
RESET(T4Timer)
RESET(T1Timer)
RESET(T2Timer)
//etofli how to set timers again ?
PhaseAltSpeech (RetVal)
RetVal
1
0
2
PhaseDisconnectLine
-
idle
PhaseB_R

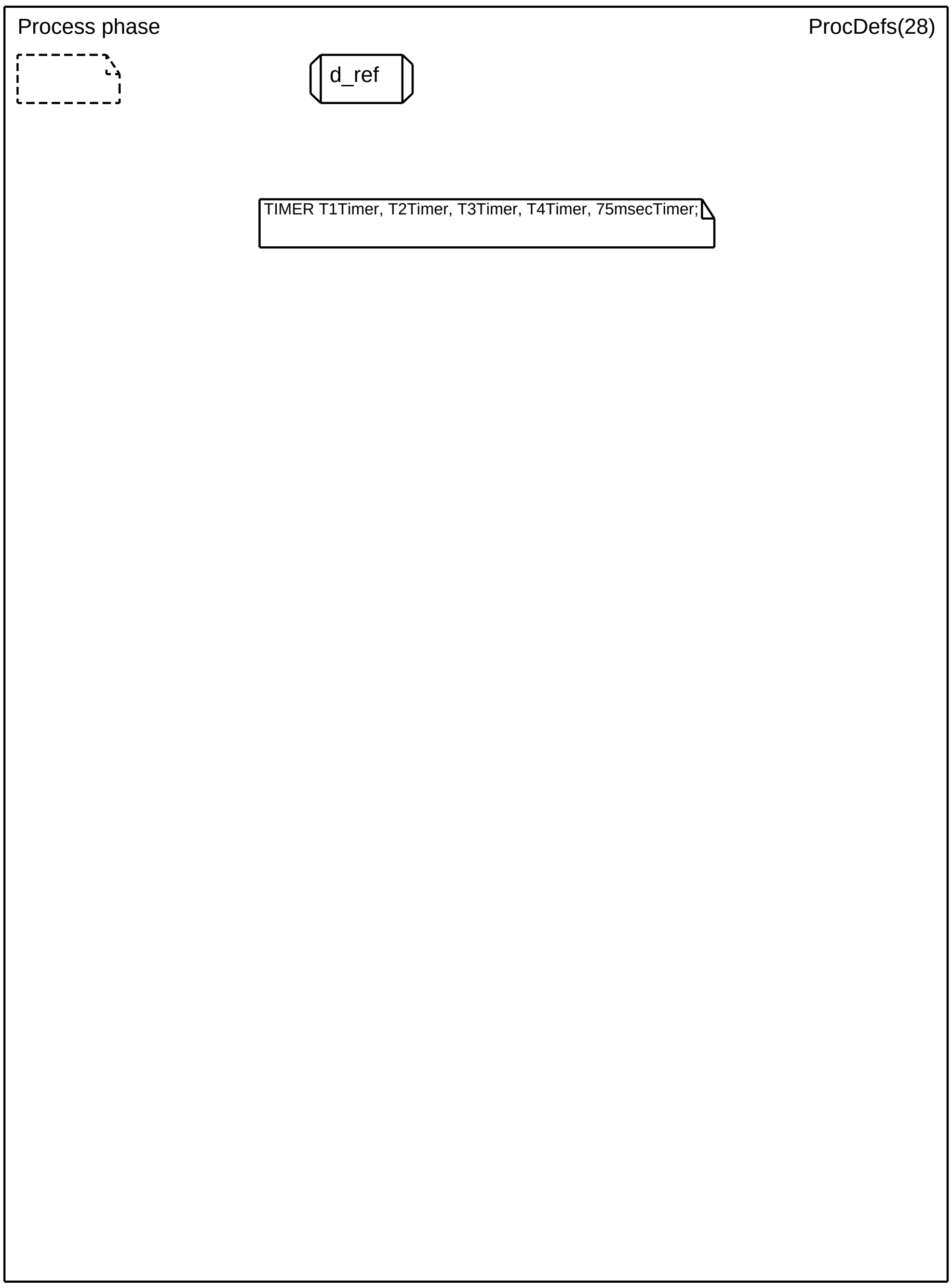

Process phase
PhaseD_T1(15)
phaseD_T
Transmitting FAX in phase D
MCF (DisData)
RTP (DisData)
RTN (DisData)
*
T4Timer
RESET(T4Timer)
RESET(T4Timer)
RESET(T4Timer)
RESET(T4Timer)
Timeouts:=Timeouts+1
OkFax via a
NokFax via a
CapableToReXmit = true
false
true
NokFax via a
SET(NOW + T4Period,T4Timer)
C_ref
C_ref
D_ref
C_ref
Timeouts=3
true
false
C_ref
NokFax via a
EOP (DisData) VIA g
NokFax via a
DCN (DisData) VIA g
phaseD_T
C_ref
C_ref
PhaseDisconnectLine
idle

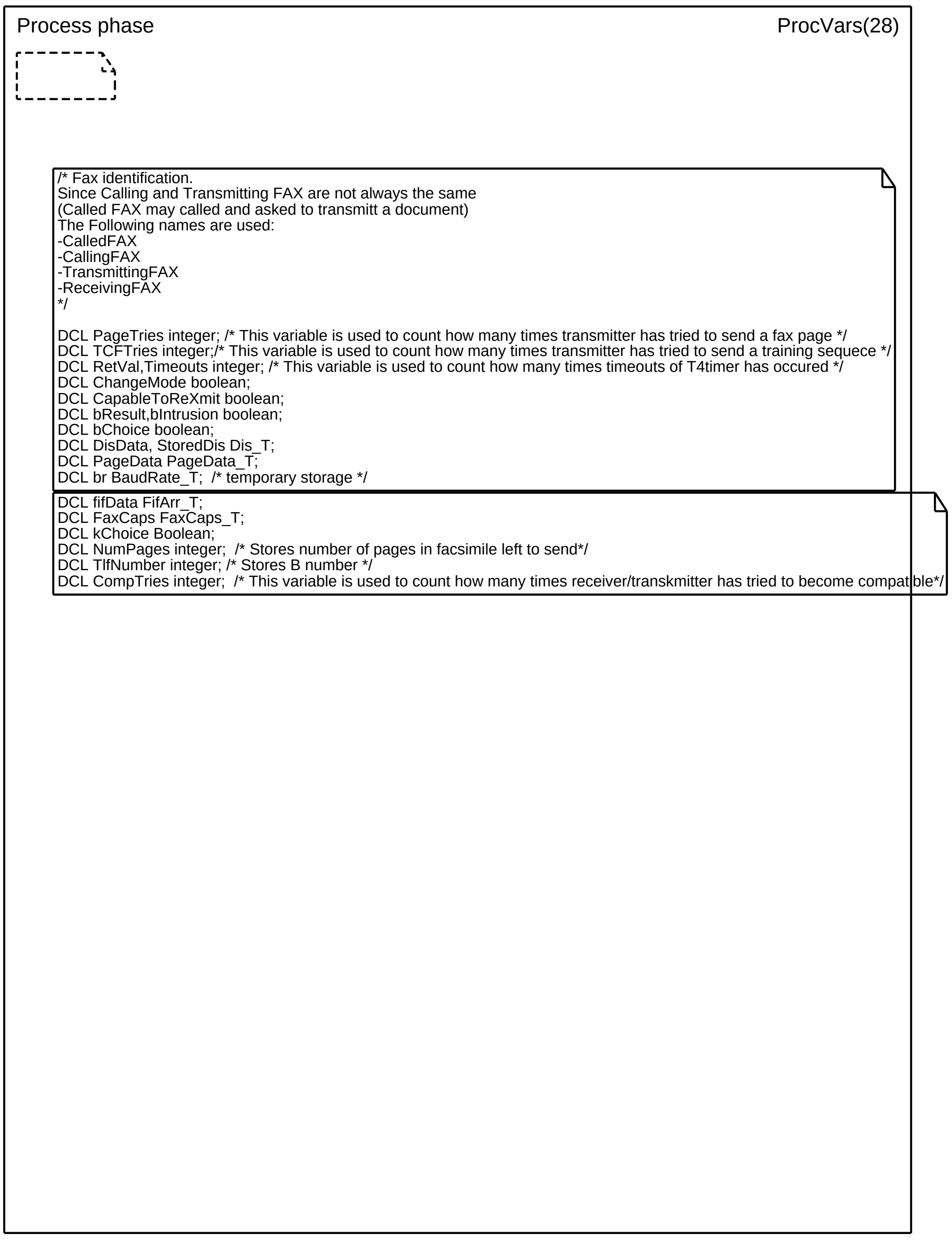
APPENDIX

D

SDL: PROCESS PHASE (TRANSFORMED)

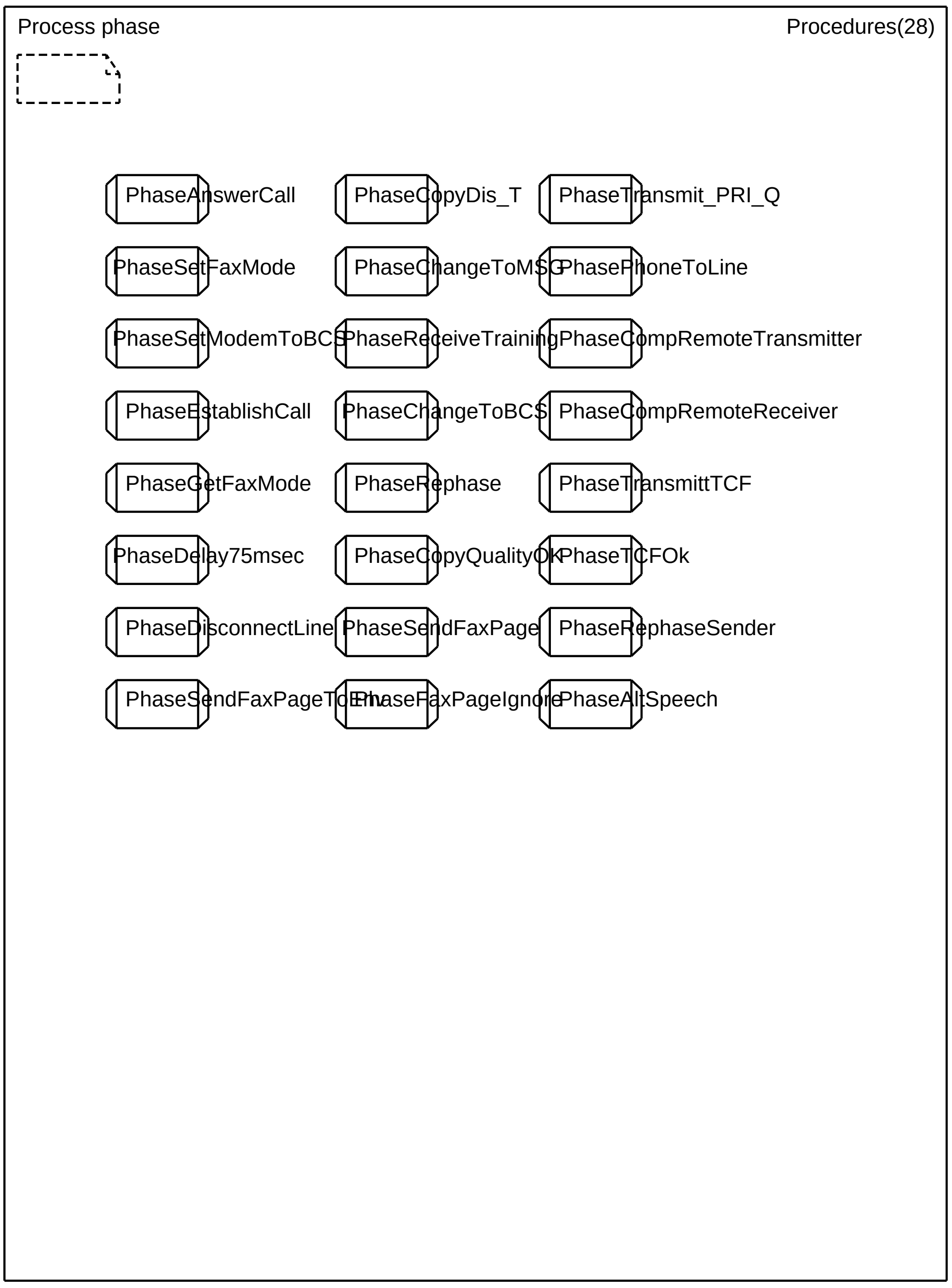
This appendix contains the SDL diagrams resulting from using transformer.pl on the diagrams in appendix C.

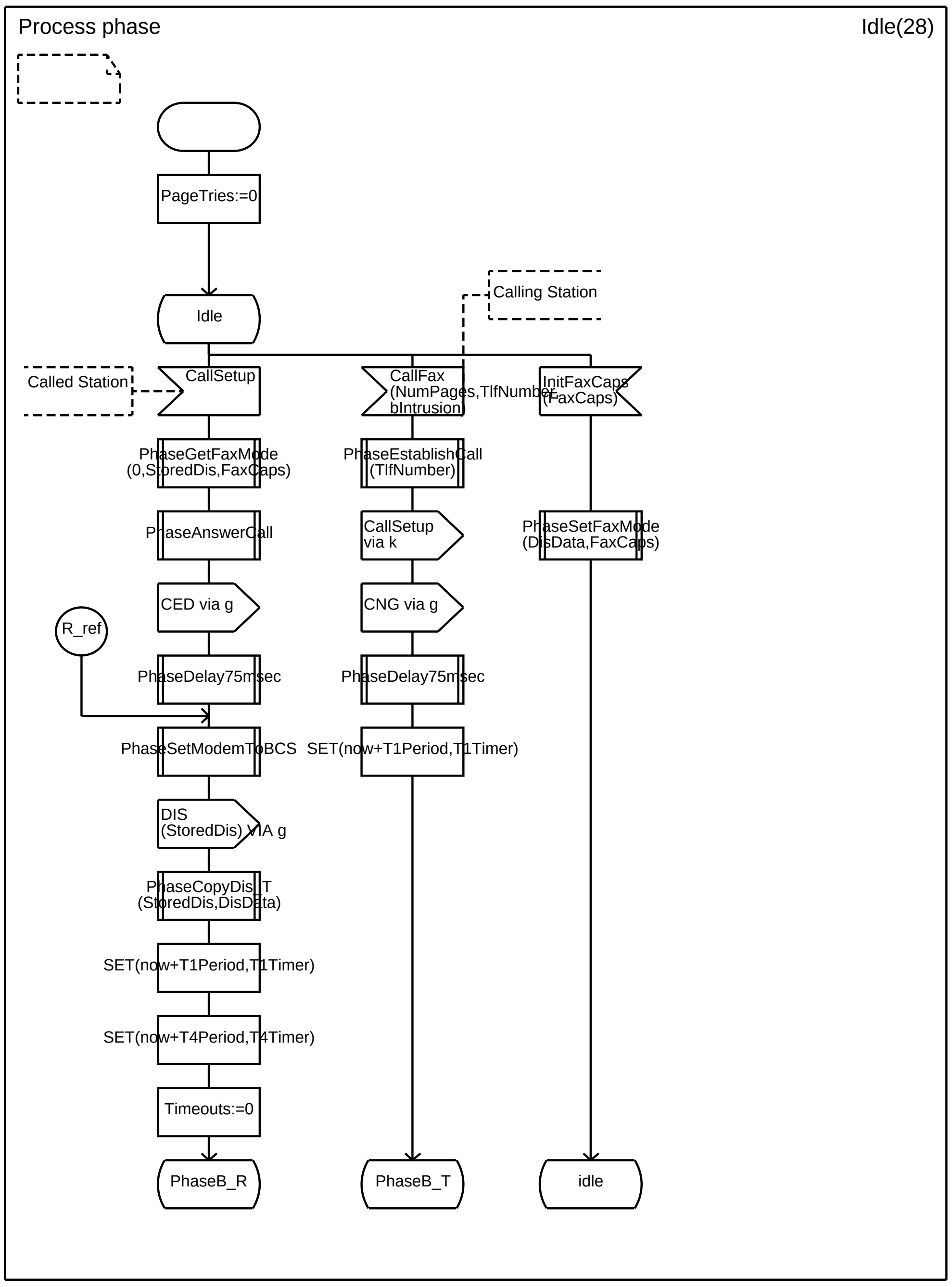

Process phase
ProcDefs(28)
d_ref
TIMER T1Timer, T2Timer, T3Timer, T4Timer, 75msecTimer;

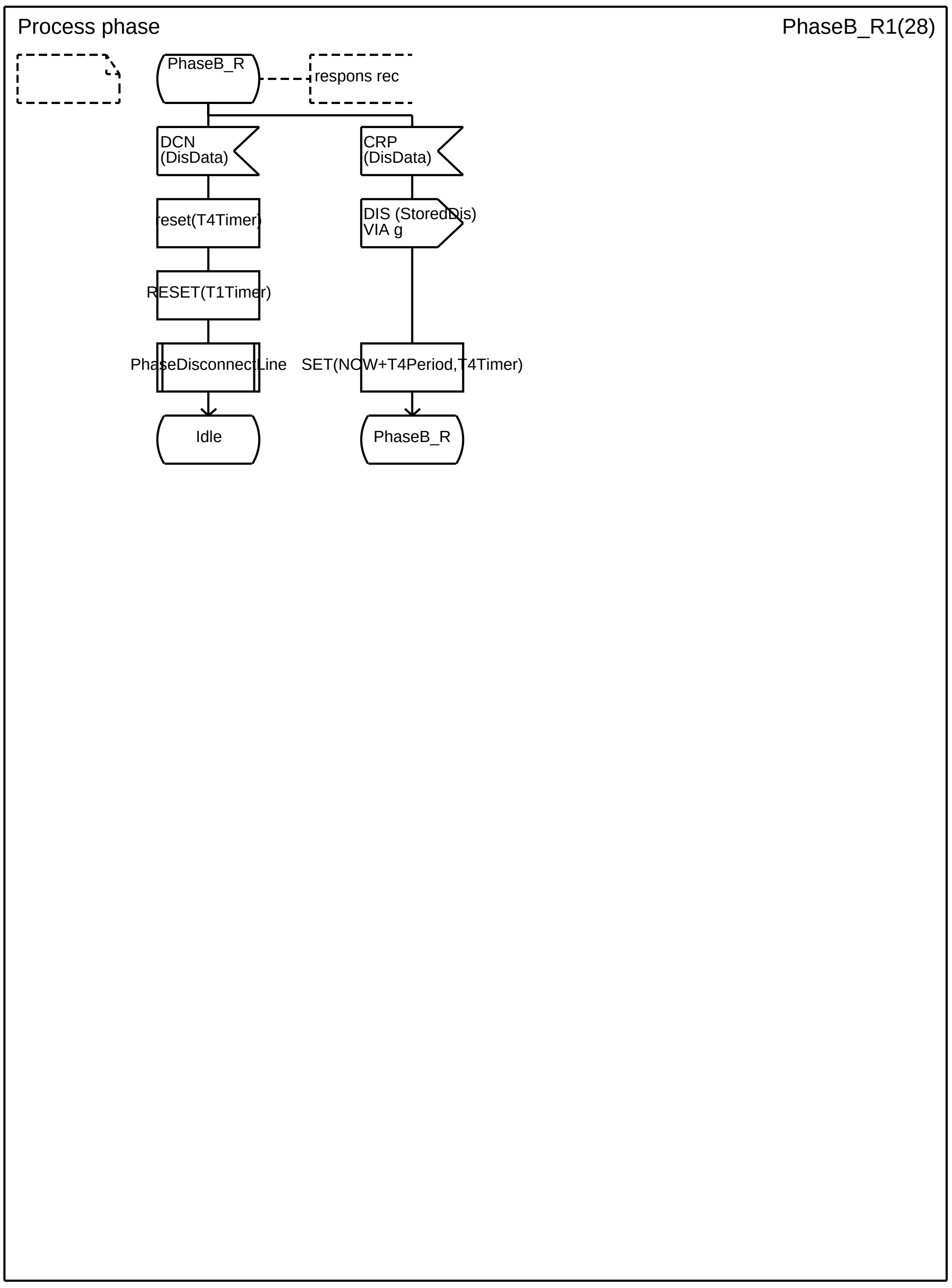


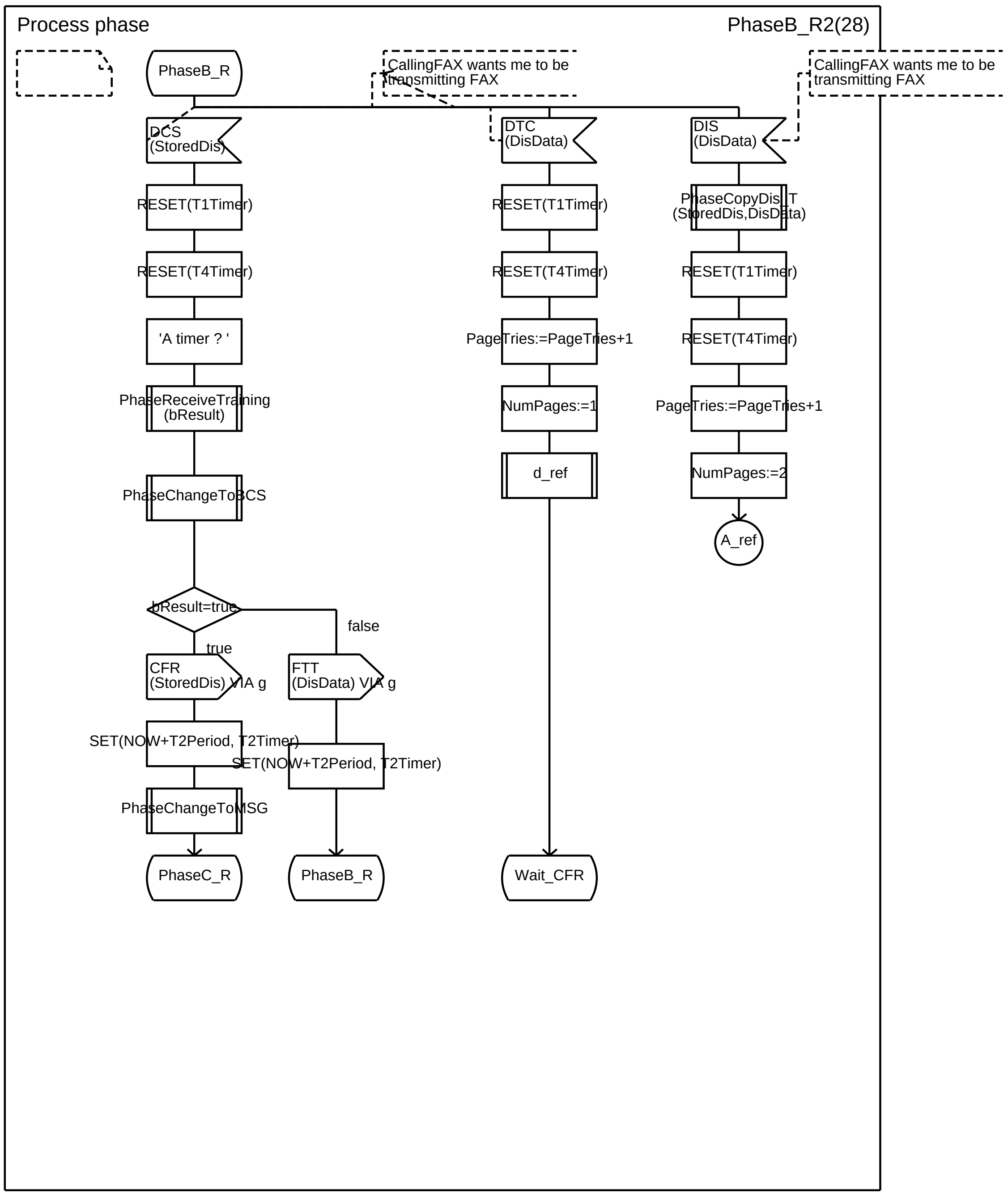
```
/* Fax identification.
Since Calling and Transmitting FAX are not always the same
(Called FAX may called and asked to transmitt a document)
The Following names are used:
-CalledFAX
-CallingFAX
-TransmittingFAX
-ReceivingFAX
*/

DCL PageTries integer; /* This variable is used to count how many times transmitter has tried to send a fax page */
DCL TCFTries integer;/* This variable is used to count how many times transmitter has tried to send a training sequece */
DCL RetVal,Timeouts integer; /* This variable is used to count how many times timeouts of T4timer has occured */
DCL ChangeMode boolean;
DCL CapableToReXmit boolean;
DCL bResult,bIntrusion boolean;
DCL bChoice boolean;
DCL DisData, StoredDis Dis_T;
DCL PageData PageData_T;
DCL br BaudRate_T;  /* temporary storage */
```

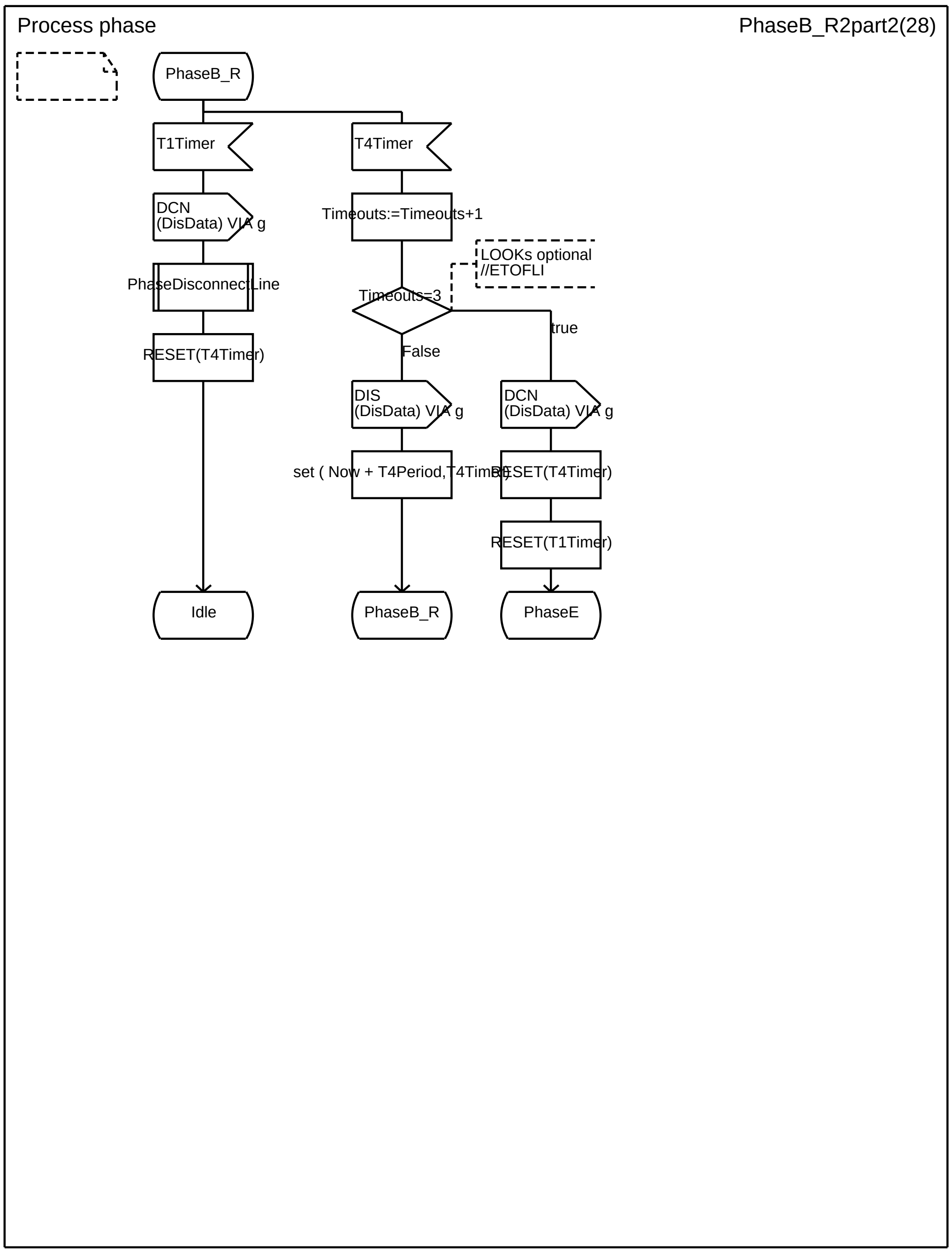
```
DCL fifData FifArr_T;
DCL FaxCaps FaxCaps_T;
DCL kChoice Boolean;
DCL NumPages integer;  /* Stores number of pages in facsimile left to send*/
DCL TlfNumber integer; /* Stores B number */
DCL CompTries integer;  /* This variable is used to count how many times receiver/transkmitter has tried to become compatible*/
```

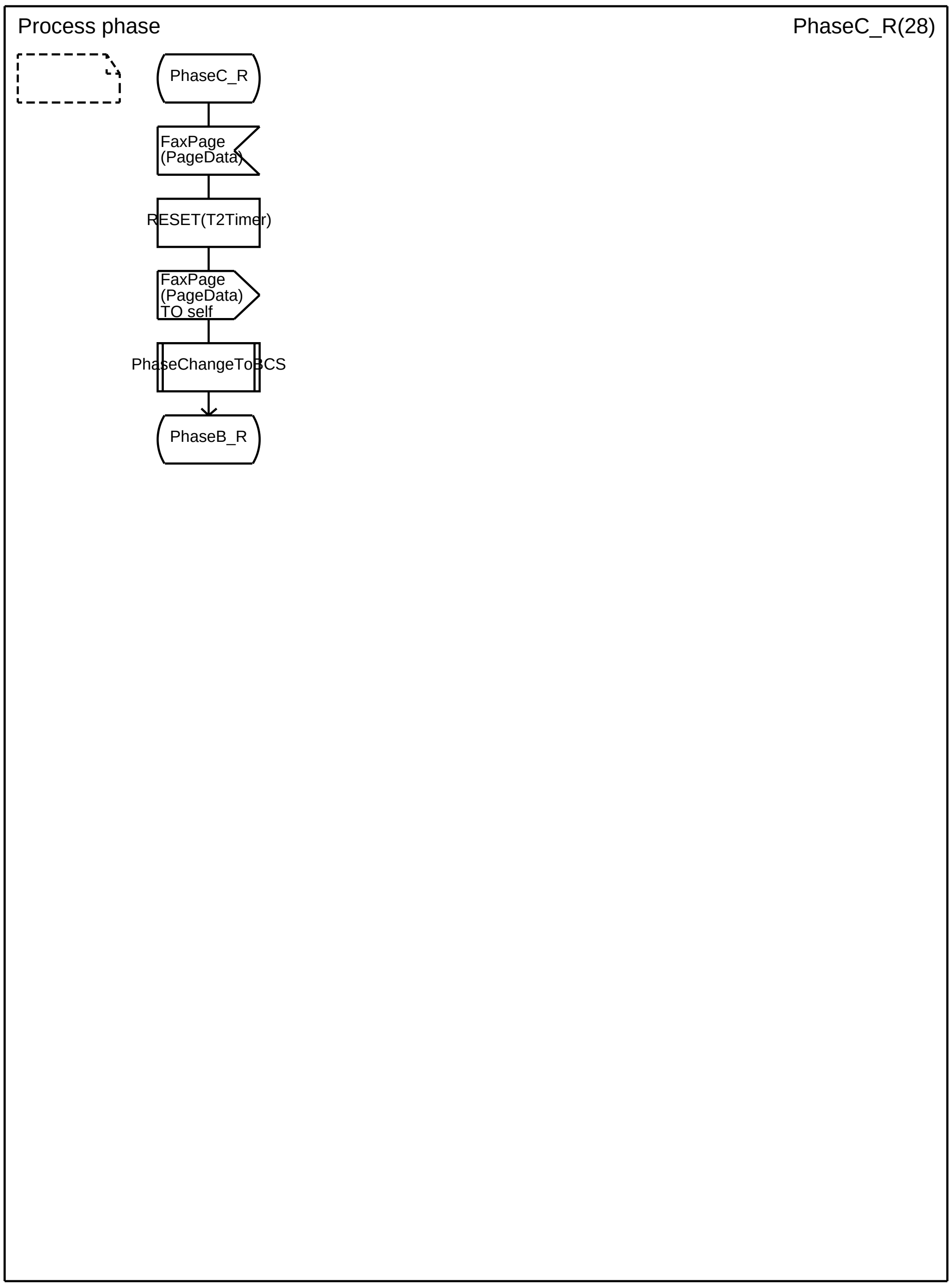

Process phase
Procedures(28)
PhaseAnswerCall
PhaseCopyDis_T
PhaseTransmit_PRI_Q
PhaseSetFaxMode
PhaseChangeToMSG
PhasePhoneToLine
PhaseSetModemToBCS
PhaseReceiveTraining
PhaseCompRemoteTransmitter
PhaseEstablishCall
PhaseChangeToBCS
PhaseCompRemoteReceiver
PhaseGetFaxMode
PhaseRephase
PhaseTransmittTCF
PhaseDelay75msec
PhaseCopyQualityOk
PhaseTCFOk
PhaseDisconnectLine
PhaseSendFaxPage
PhaseRephaseSender
PhaseSendFaxPageToEh
PhaseFaxPageIgnore
PhaseAltSpeech

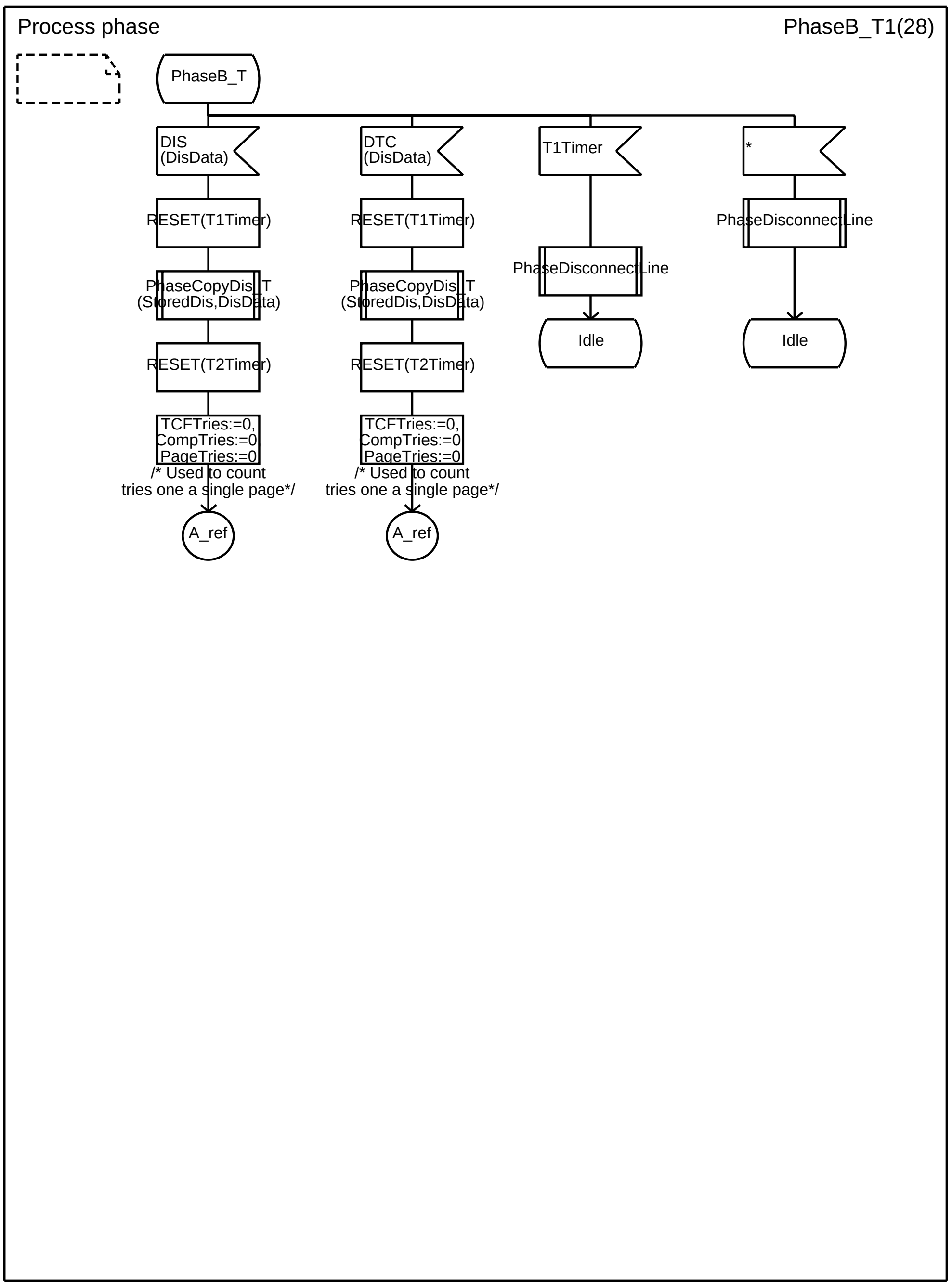

Process phase
Idle(28)
PageTries:=0
Idle
Calling Station
Called Station
CallSetup
CallFax (NumPages,TlfNumber, bIntrusion)
InitFaxCaps (FaxCaps)
PhaseGetFaxMode (0,StoredDis,FaxCaps)
PhaseEstablishCall (TlfNumber)
PhaseAnswerCall
CallSetup via k
PhaseSetFaxMode (DisData,FaxCaps)
CED via g
CNG via g
R_ref
PhaseDelay75msec
PhaseDelay75msec
PhaseSetModemToBCS
SET(now+T1Period,T1Timer)
DIS (StoredDis) VIA g
PhaseCopyDis_T (StoredDis,DisData)
SET(now+T1Period,T1Timer)
SET(now+T4Period,T4Timer)
Timeouts:=0
PhaseB_R
PhaseB_T
idle

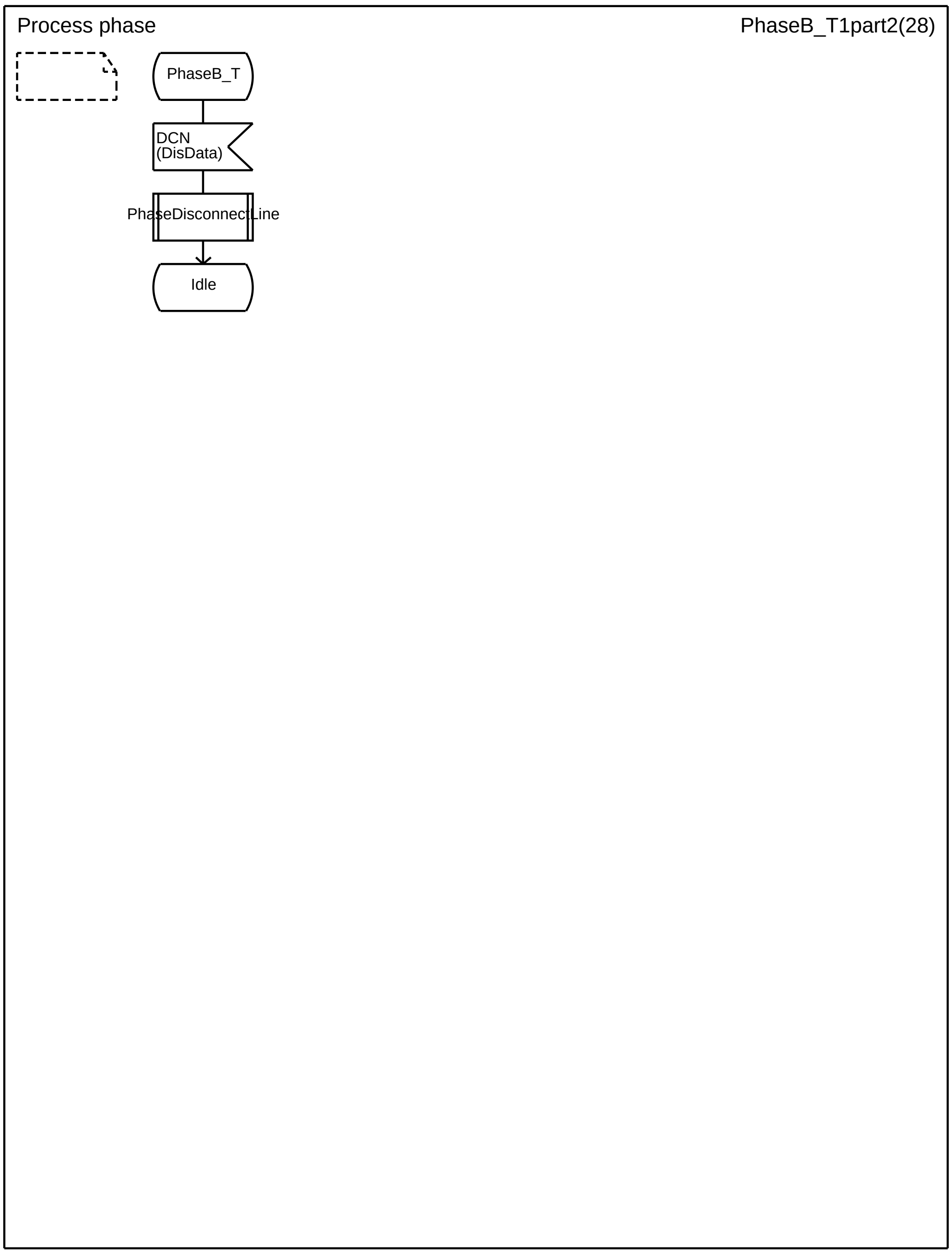

Process phase
PhaseB_R1(28)
PhaseB_R
respons rec
DCN (DisData)
CRP (DisData)
reset(T4Timer)
DIS (StoredDis) VIA g
RESET(T1Timer)
PhaseDisconnectLine
SET(NOW+T4Period,T4Timer)
Idle
PhaseB_R

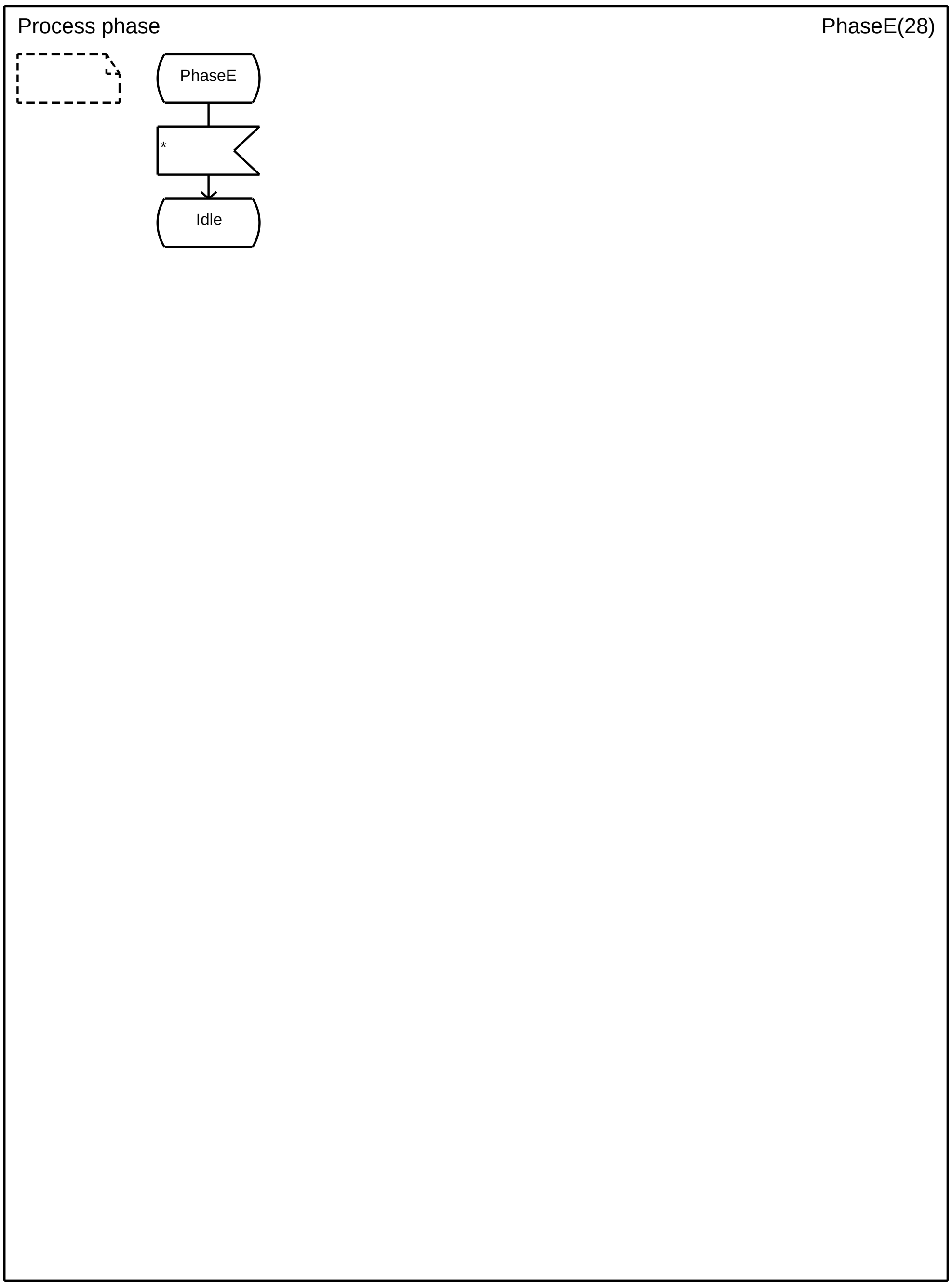

Process phase
PhaseB_R2(28)
PhaseB_R
CallingFAX wants me to be transmitting FAX
CallingFAX wants me to be transmitting FAX
DCS (StoredDis)
DTC (DisData)
DIS (DisData)
RESET(T1Timer)
RESET(T4Timer)
'A timer ? '
PhaseReceiveTraining (bResult)
PhaseChangeToDCS
bResult=true
true
false
CFR (StoredDis) VIA g
FTT (DisData) VIA g
SET(NOW+T2Period, T2Timer)
SET(NOW+T2Period, T2Timer)
PhaseChangeToMSG
PhaseC_R
PhaseB_R
RESET(T1Timer)
RESET(T4Timer)
PageTries:=PageTries+1
NumPages:=1
d_ref
Wait_CFR
PhaseCopyDis_T (StoredDis,DisData)
RESET(T1Timer)
RESET(T4Timer)
PageTries:=PageTries+1
NumPages:=2
A_ref

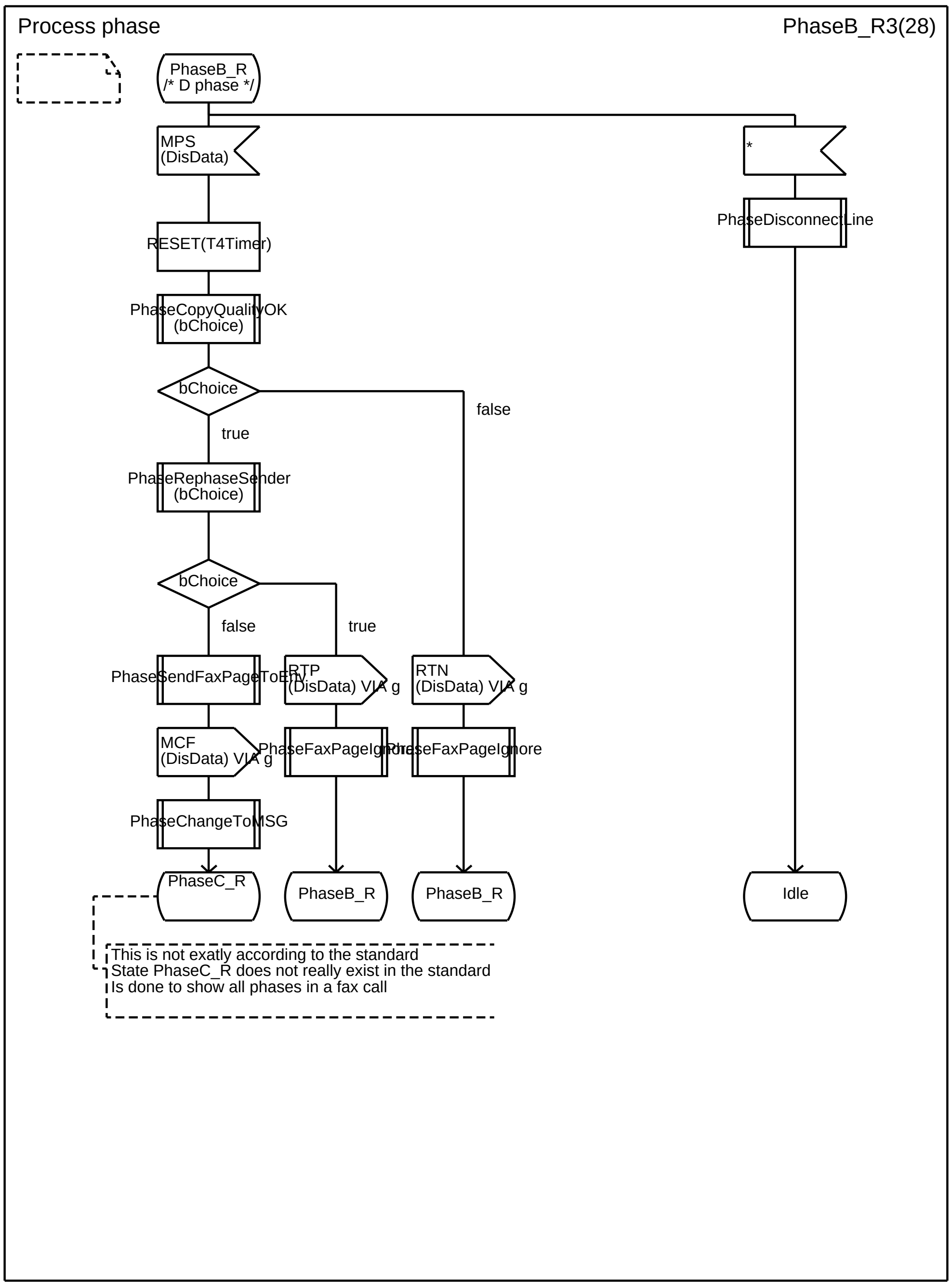

Process phase
PhaseB_R2part2(28)
PhaseB_R
T1Timer
T4Timer
DCN
(DisData) VIA g
Timeouts:=Timeouts+1
LOOKs optional
//ETOFLI
PhaseDisconnectLine
Timeouts=3
true
RESET(T4Timer)
False
DIS
(DisData) VIA g
DCN
(DisData) VIA g
set (Now + T4Period,T4Timer)
RESET(T4Timer)
RESET(T1Timer)
Idle
PhaseB_R
PhaseE

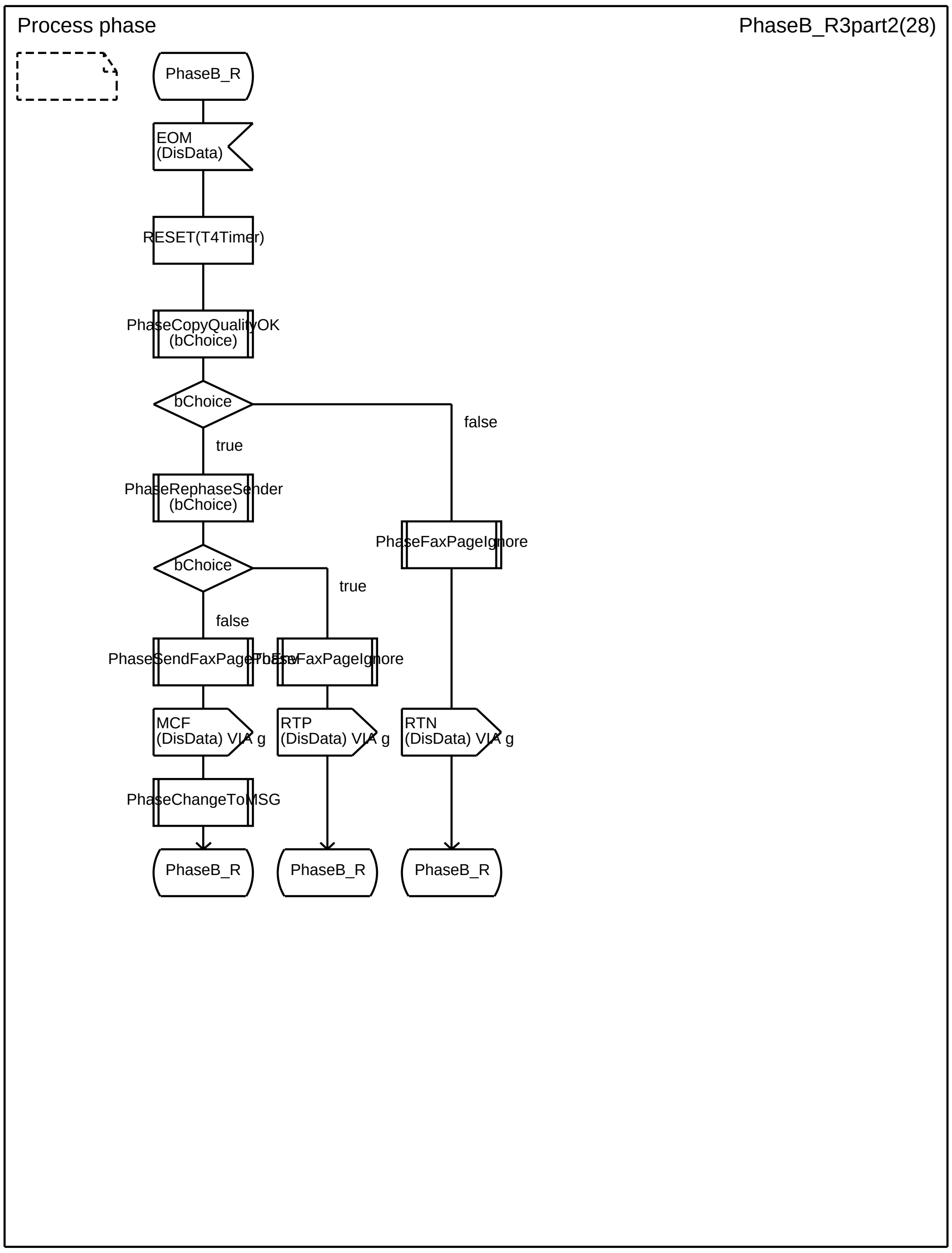

Process phase
PhaseC_R(28)
PhaseC_R
FaxPage (PageData)
RESET(T2Timer)
FaxPage (PageData) TO self
PhaseChangeToBCS
PhaseB_R

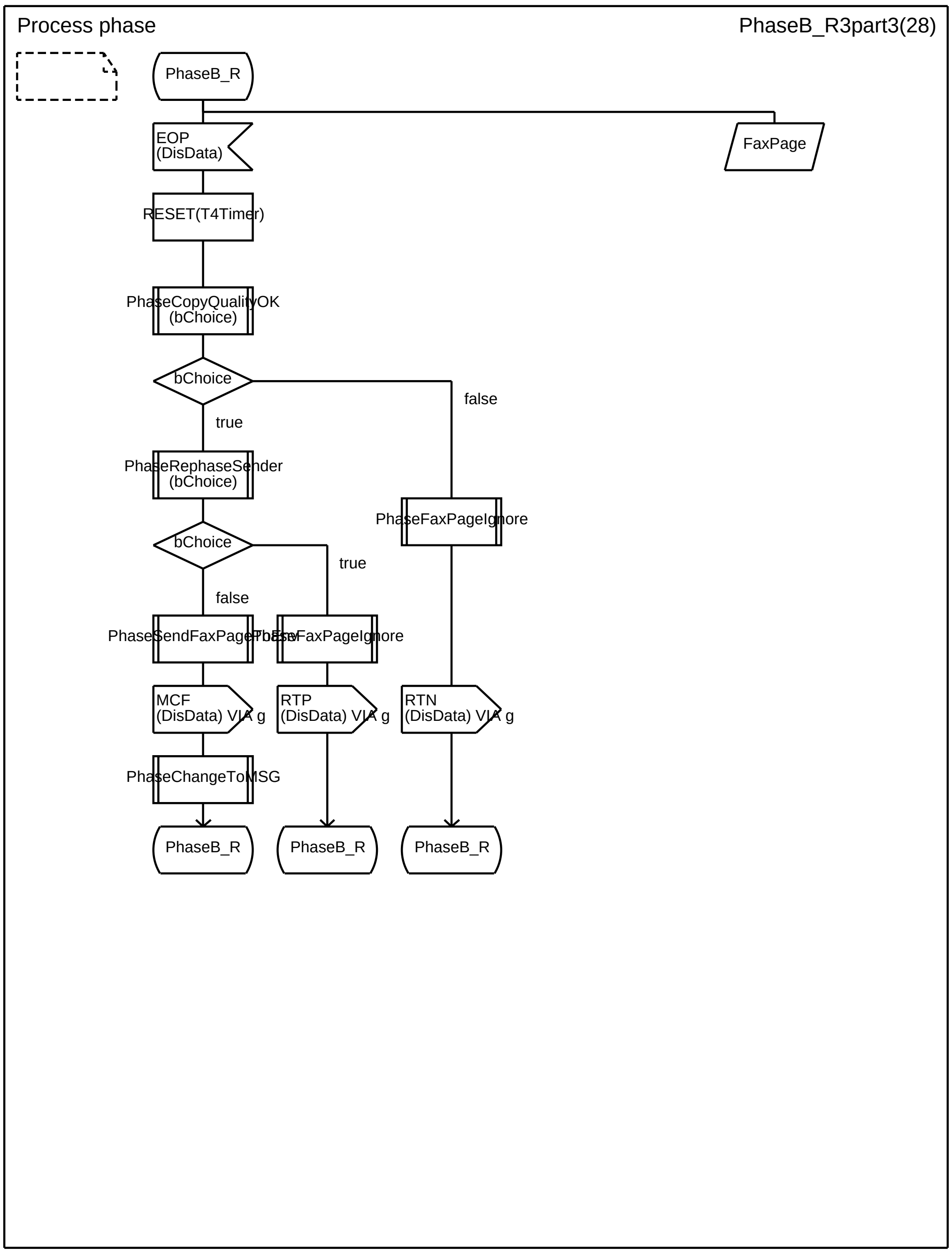

Process phase
PhaseB_T1(28)
PhaseB_T
DIS (DisData)
DTC (DisData)
T1Timer
*
RESET(T1Timer)
RESET(T1Timer)
PhaseDisconnectLine
PhaseDisconnectLine
PhaseCopyDis_T (StoredDis,DisData)
PhaseCopyDis_T (StoredDis,DisData)
Idle
Idle
RESET(T2Timer)
RESET(T2Timer)
TCFTries:=0, CompTries:=0 PageTries:=0
/* Used to count tries one a single page*/
TCFTries:=0, CompTries:=0 PageTries:=0
/* Used to count tries one a single page*/
A_ref
A_ref

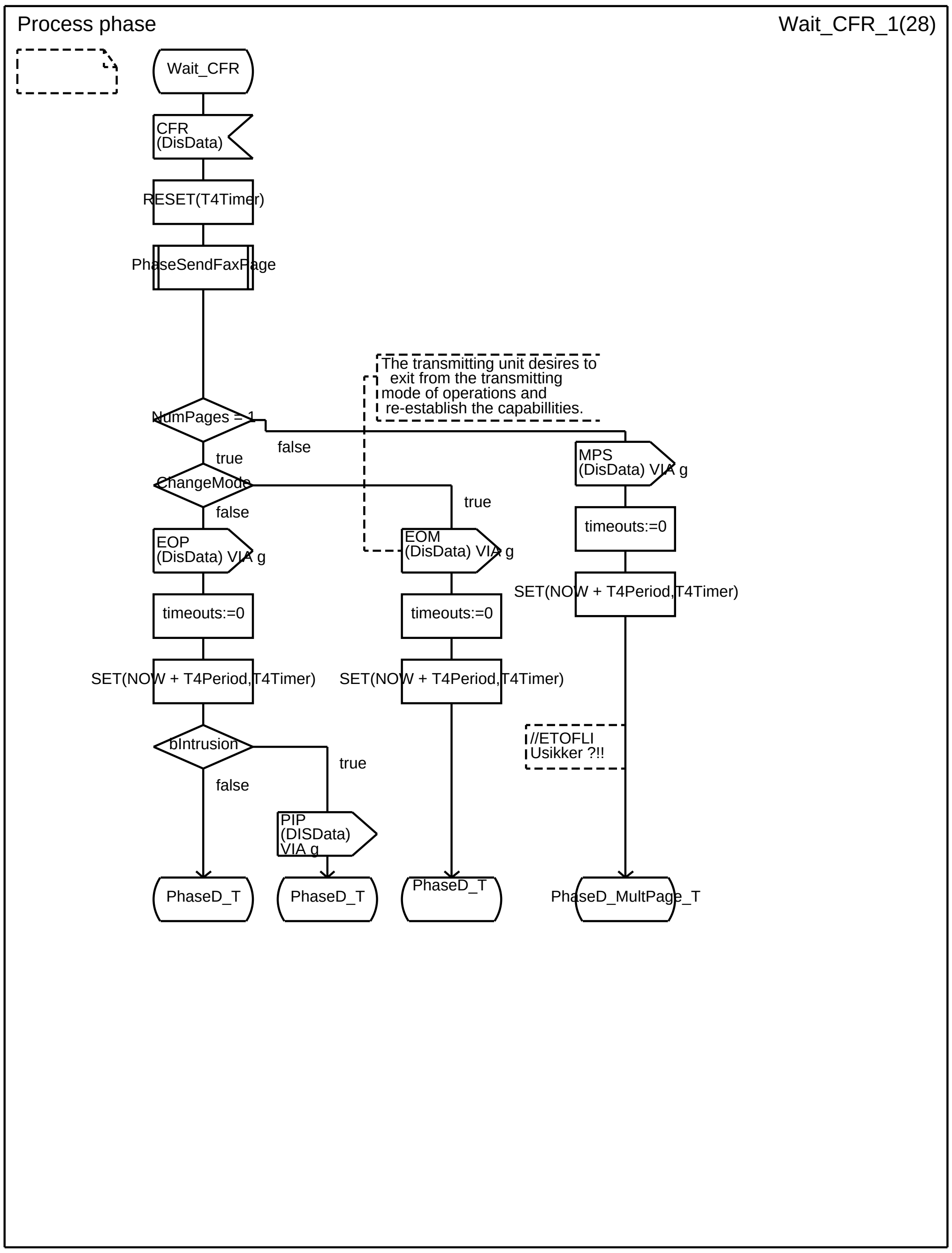

Process phase
PhaseB_T1part2(28)
PhaseB_T
DCN
(DisData)
PhaseDisconnectLine
Idle

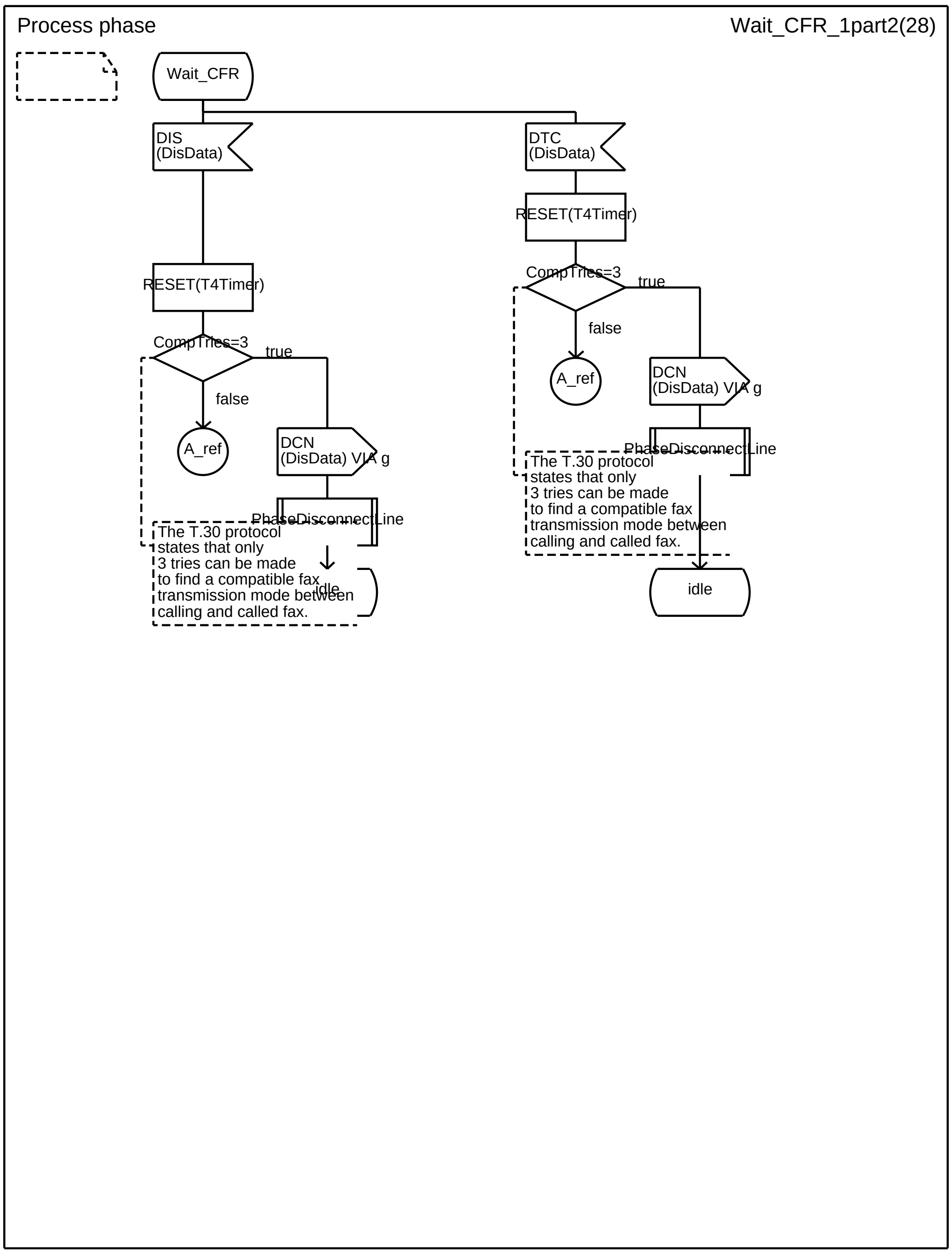

Process phase
PhaseE(28)
PhaseE
*
Idle

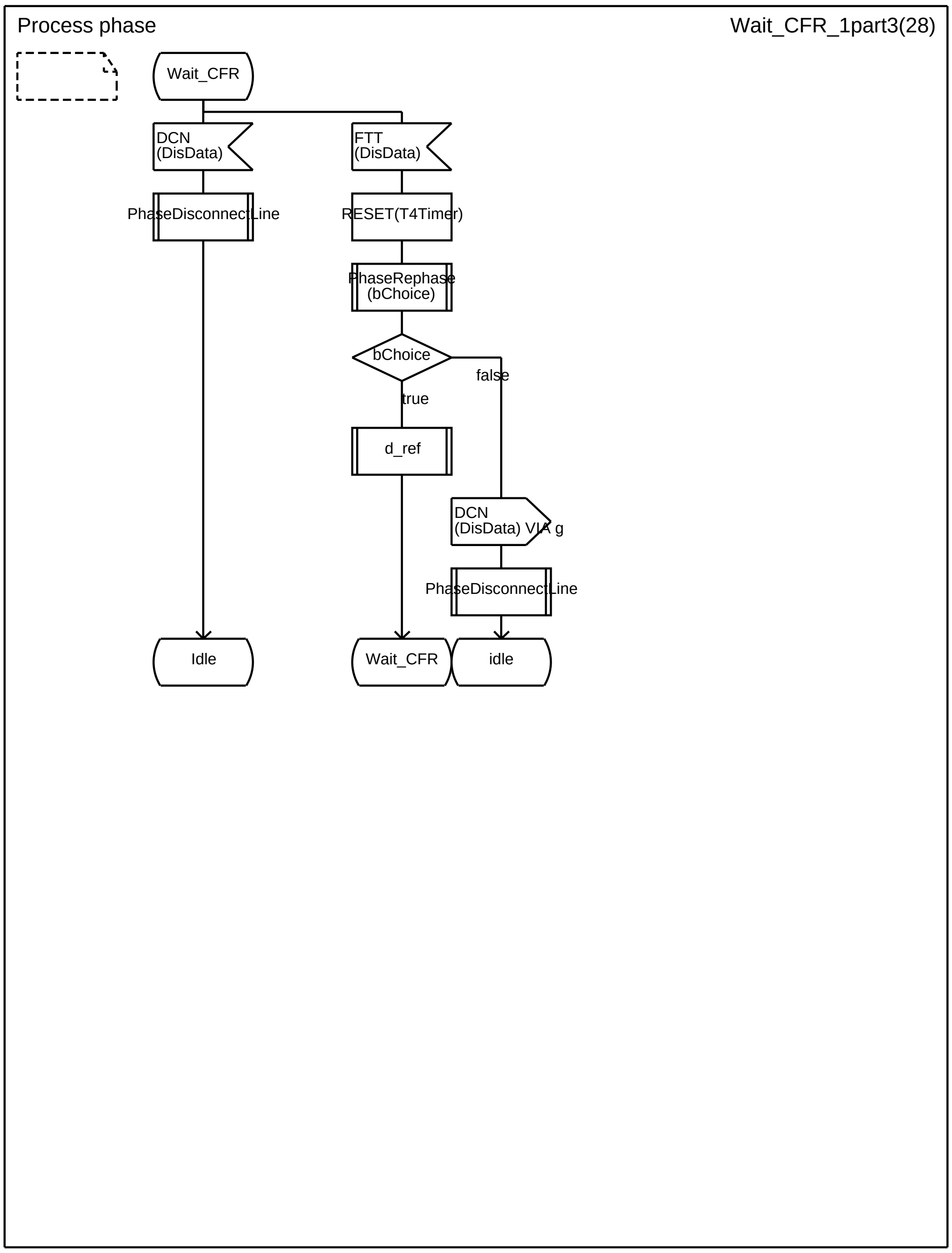

Process phase
PhaseB_R3(28)
PhaseB_R
/* D phase */
MPS
(DisData)
*
PhaseDisconnectLine
RESET(T4Timer)
PhaseCopyQualityOK
(bChoice)
bChoice
false
true
PhaseRephaseSender
(bChoice)
bChoice
false
true
PhaseSendFaxPageToEnv
RTP
(DisData) VIA g
RTN
(DisData) VIA g
MCF
(DisData) VIA g
PhaseFaxPageIgnore
PhaseFaxPageIgnore
PhaseChangeToMSG
PhaseC_R
PhaseB_R
PhaseB_R
Idle
This is not exatly according to the standard
State PhaseC_R does not really exist in the standard
Is done to show all phases in a fax call

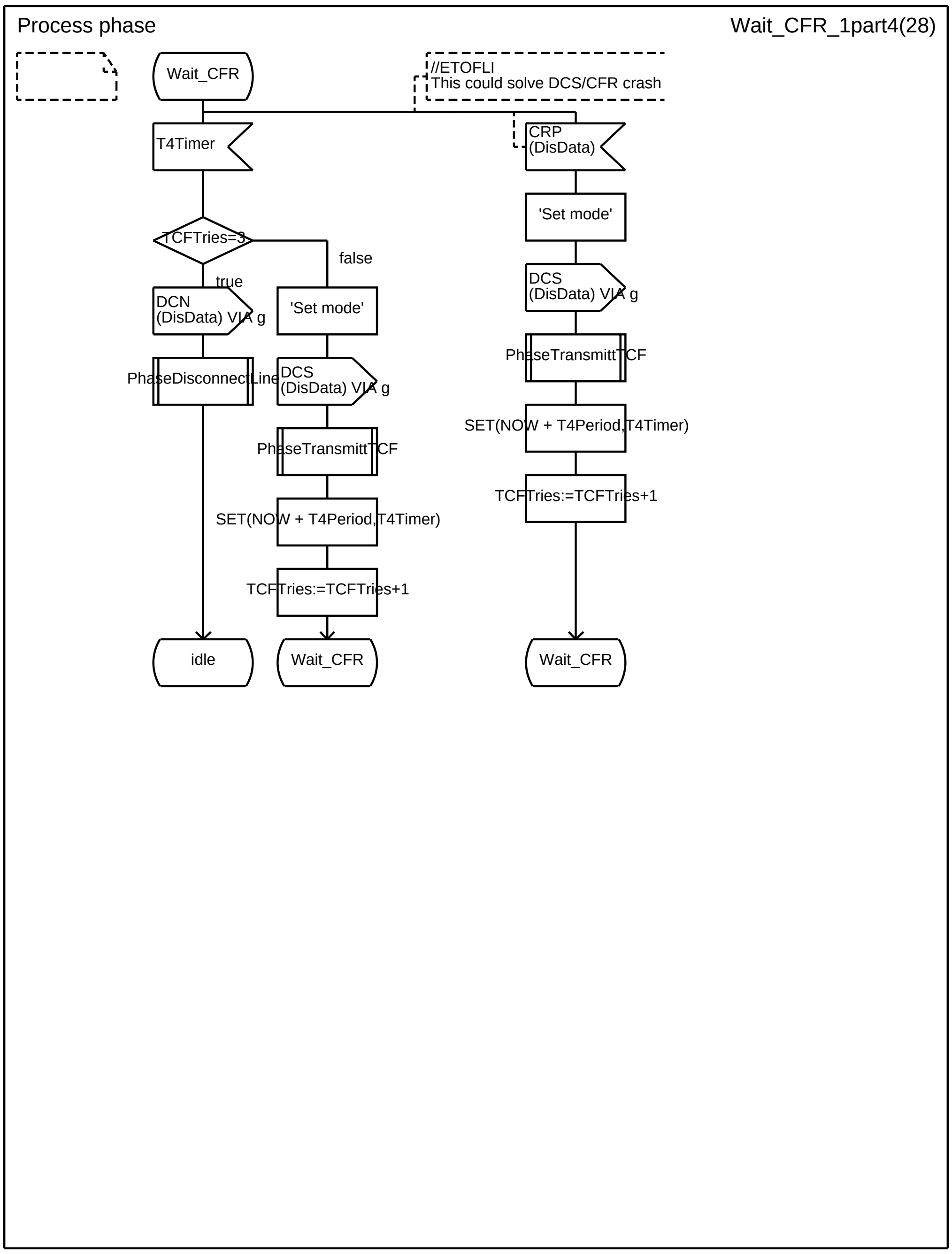

Process phase
PhaseB_R3part2(28)
PhaseB_R
EOM
(DisData)
RESET(T4Timer)
PhaseCopyQualityOK
(bChoice)
bChoice
false
true
PhaseRephaseSender
(bChoice)
PhaseFaxPageIgnore
bChoice
true
false
PhaseSendFaxPageToEsu
PhaseFaxPageIgnore
MCF
(DisData) VIA g
RTP
(DisData) VIA g
RTN
(DisData) VIA g
PhaseChangeToMSG
PhaseB_R
PhaseB_R
PhaseB_R

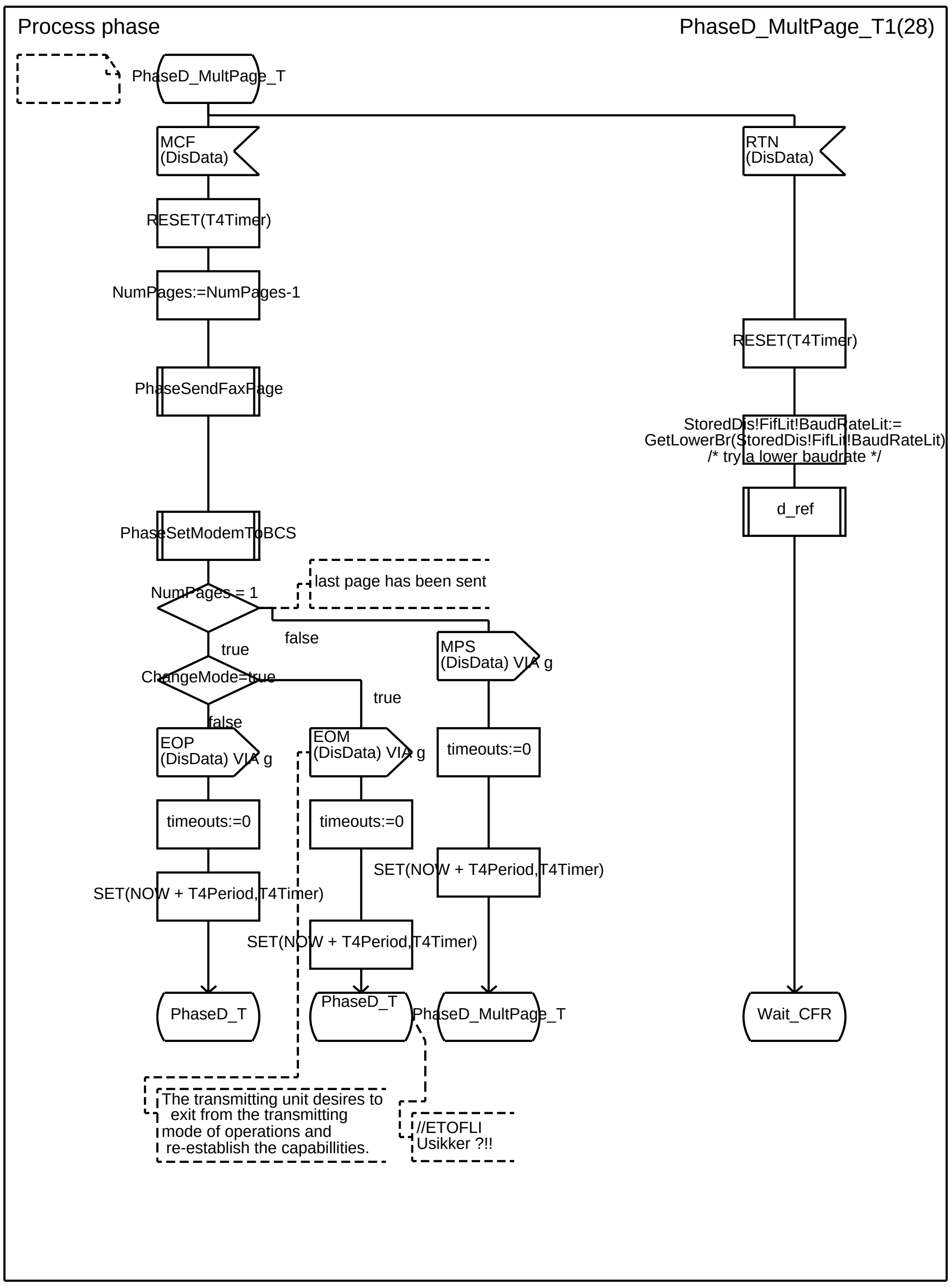

Process phase
PhaseB_R3part3(28)
PhaseB_R
EOP
(DisData)
FaxPage
RESET(T4Timer)
PhaseCopyQualityOK
(bChoice)
bChoice
false
true
PhaseRephaseSender
(bChoice)
PhaseFaxPageIgnore
bChoice
true
false
PhaseSendFaxPageToBsw
PhaseFaxPageIgnore
MCF
(DisData) VIA g
RTP
(DisData) VIA g
RTN
(DisData) VIA g
PhaseChangeToMSG
PhaseB_R
PhaseB_R
PhaseB_R

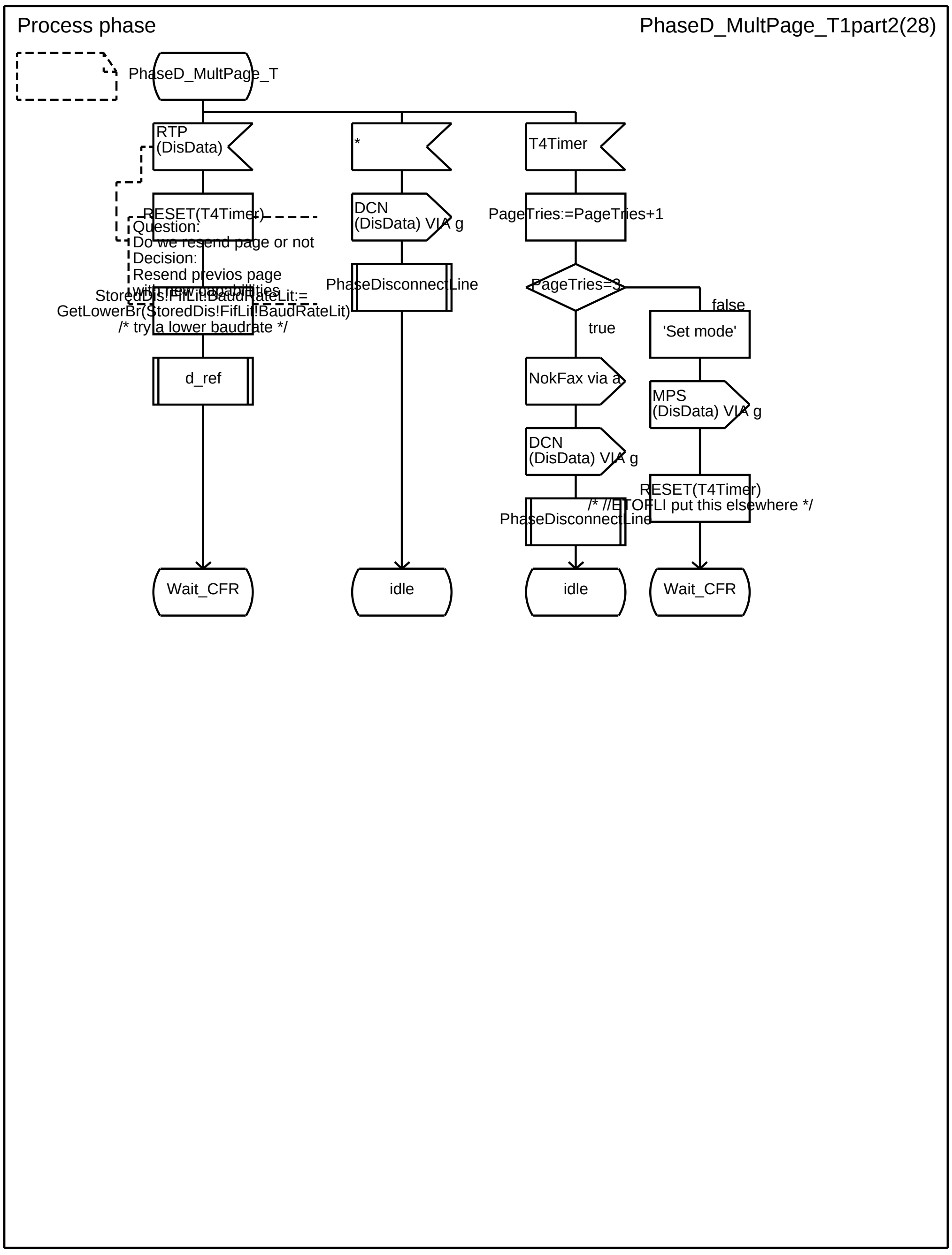

Process phase
Wait_CFR_1(28)
Wait_CFR
CFR
(DisData)
RESET(T4Timer)
PhaseSendFaxPage
The transmitting unit desires to
exit from the transmitting
mode of operations and
re-establish the capabillities.
NumPages = 1
false
true
ChangeMode
true
false
MPS
(DisData) VIA g
timeouts:=0
SET(NOW + T4Period,T4Timer)
EOP
(DisData) VIA g
EOM
(DisData) VIA g
timeouts:=0
timeouts:=0
SET(NOW + T4Period,T4Timer)
SET(NOW + T4Period,T4Timer)
//ETOFLI
Usikker ?!!
bIntrusion
true
false
PIP
(DISData)
VIA g
PhaseD_T
PhaseD_T
PhaseD_T
PhaseD_MultPage_T

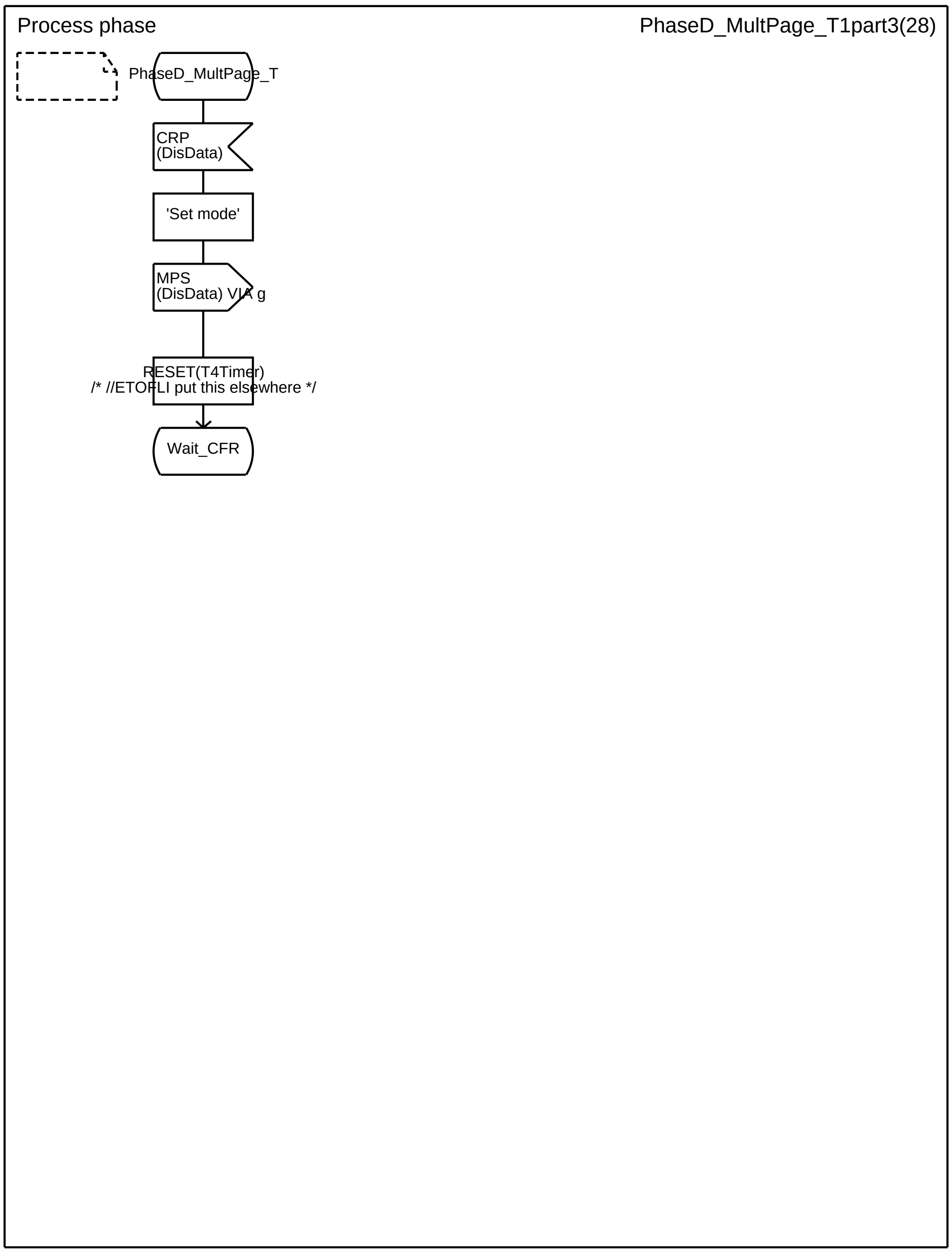

Process phase
Wait_CFR_1part2(28)
Wait_CFR
DIS (DisData)
DTC (DisData)
RESET(T4Timer)
RESET(T4Timer)
CompTries=3
true
false
A_ref
DCN (DisData) VIA g
PhaseDisconnectLine
The T.30 protocol states that only 3 tries can be made to find a compatible fax transmission mode between calling and called fax.
idle
CompTries=3
true
false
A_ref
DCN (DisData) VIA g
PhaseDisconnectLine
The T.30 protocol states that only 3 tries can be made to find a compatible fax transmission mode between calling and called fax.
idle

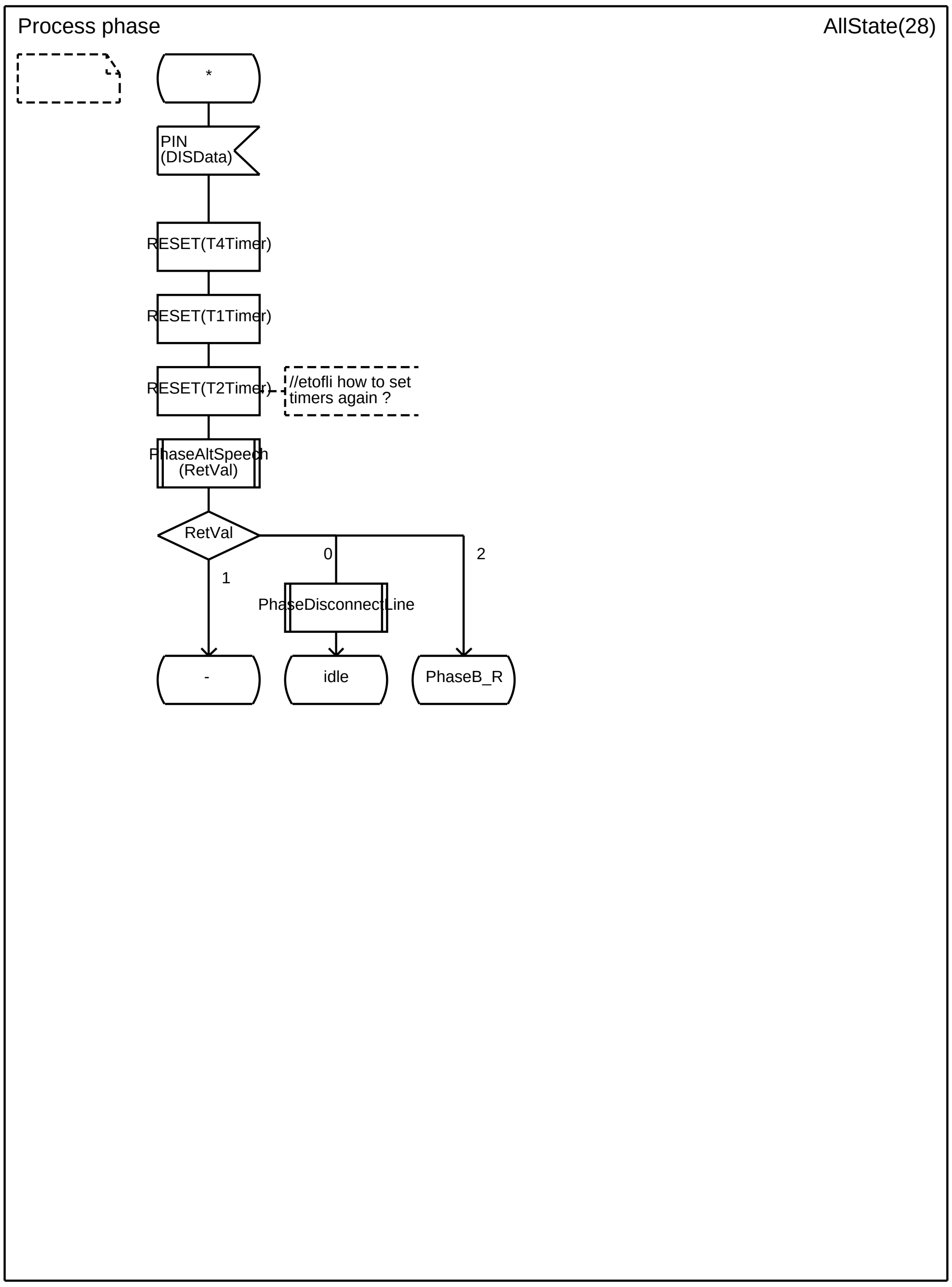

Process phase
Wait_CFR_1part3(28)
Wait_CFR
DCN
(DisData)
FTT
(DisData)
PhaseDisconnectLine
RESET(T4Timer)
PhaseRephase
(bChoice)
bChoice
false
true
d_ref
DCN
(DisData) VIA g
PhaseDisconnectLine
Idle
Wait_CFR
idle

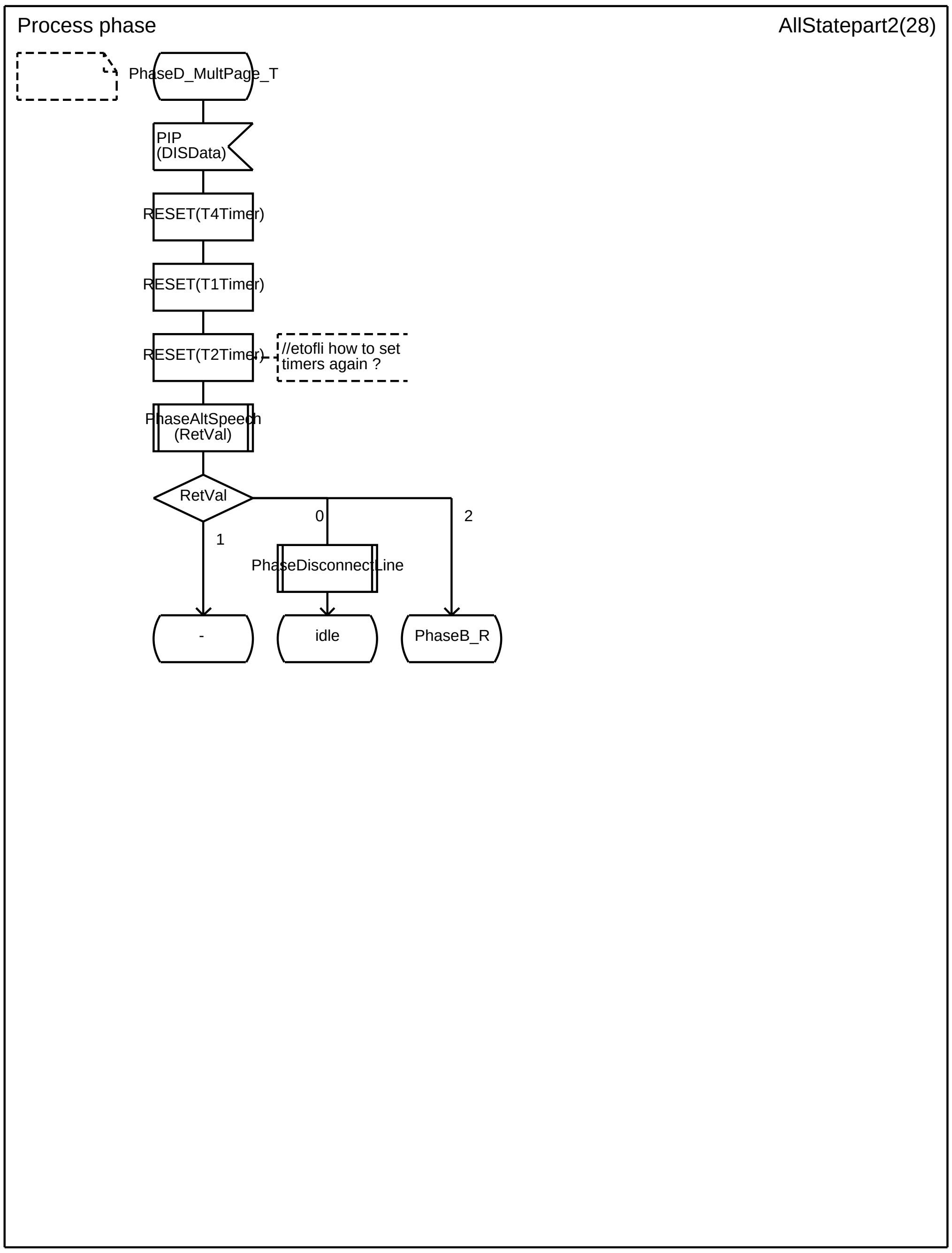

Process phase
Wait_CFR_1part4(28)
Wait_CFR
//ETOFLI
This could solve DCS/CFR crash
T4Timer
CRP
(DisData)
'Set mode'
TCFTries=3
false
true
DCN
(DisData) VIA g
'Set mode'
DCS
(DisData) VIA g
PhaseTransmittTCF
PhaseDisconnectLine
DCS
(DisData) VIA g
SET(NOW + T4Period,T4Timer)
PhaseTransmittTCF
TCFTries:=TCFTries+1
SET(NOW + T4Period,T4Timer)
TCFTries:=TCFTries+1
idle
Wait_CFR
Wait_CFR

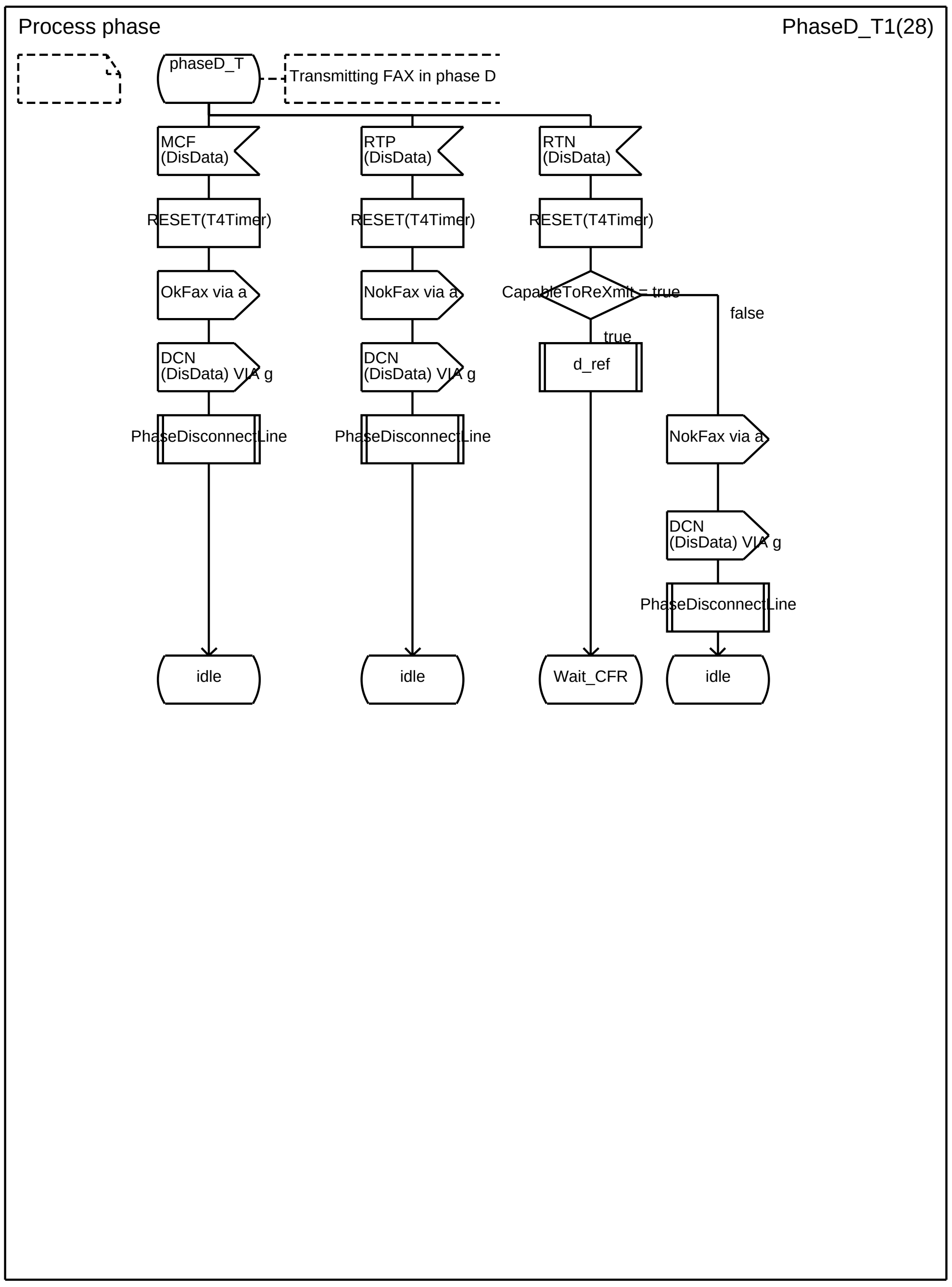

Process phase
PhaseD_MultPage_T1(28)
PhaseD_MultPage_T
MCF
(DisData)
RTN
(DisData)
RESET(T4Timer)
NumPages:=NumPages-1
PhaseSendFaxPage
RESET(T4Timer)
StoredDis!FifLit!BaudRateLit:=
GetLowerBr(StoredDis!FifLit!BaudRateLit)
/* try a lower baudrate */
d_ref
PhaseSetModemToBCS
last page has been sent
NumPages = 1
true
false
MPS
(DisData) VIA g
ChangeMode=true
true
false
EOP
(DisData) VIA g
EOM
(DisData) VIA g
timeouts:=0
timeouts:=0
timeouts:=0
SET(NOW + T4Period,T4Timer)
SET(NOW + T4Period,T4Timer)
SET(NOW + T4Period,T4Timer)
PhaseD_T
PhaseD_T
PhaseD_MultPage_T
Wait_CFR
The transmitting unit desires to
exit from the transmitting
mode of operations and
re-establish the capabillities.
//ETOFLI
Usikker ?!!

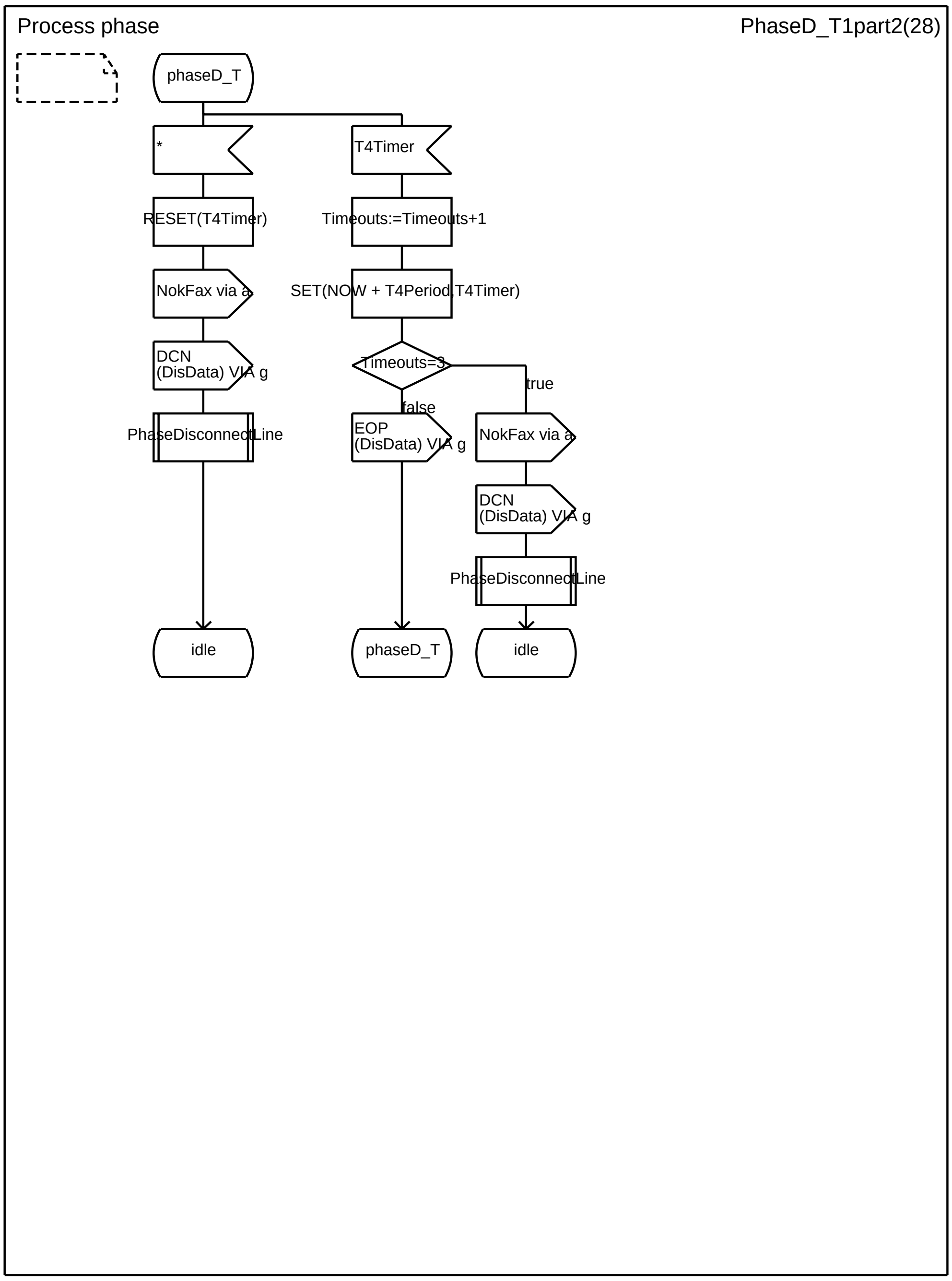

Process phase
PhaseD_MultPage_T1part2(28)
PhaseD_MultPage_T
RTP
(DisData)
*
T4Timer
RESET(T4Timer)
Question:
Do we resend page or not
Decision:
Resend previos page
with new capabilities
StoredDis!FifLit!BaudRateLit:=
GetLowerBr(StoredDis!FifLit!BaudRateLit)
/* try a lower baudrate */
DCN
(DisData) VIA g
PhaseDisconnectLine
PageTries:=PageTries+1
PageTries=3
true
false
'Set mode'
d_ref
NokFax via a
MPS
(DisData) VIA g
DCN
(DisData) VIA g
RESET(T4Timer)
/* //ETOFLI put this elsewhere */
PhaseDisconnectLine
Wait_CFR
idle
idle
Wait_CFR

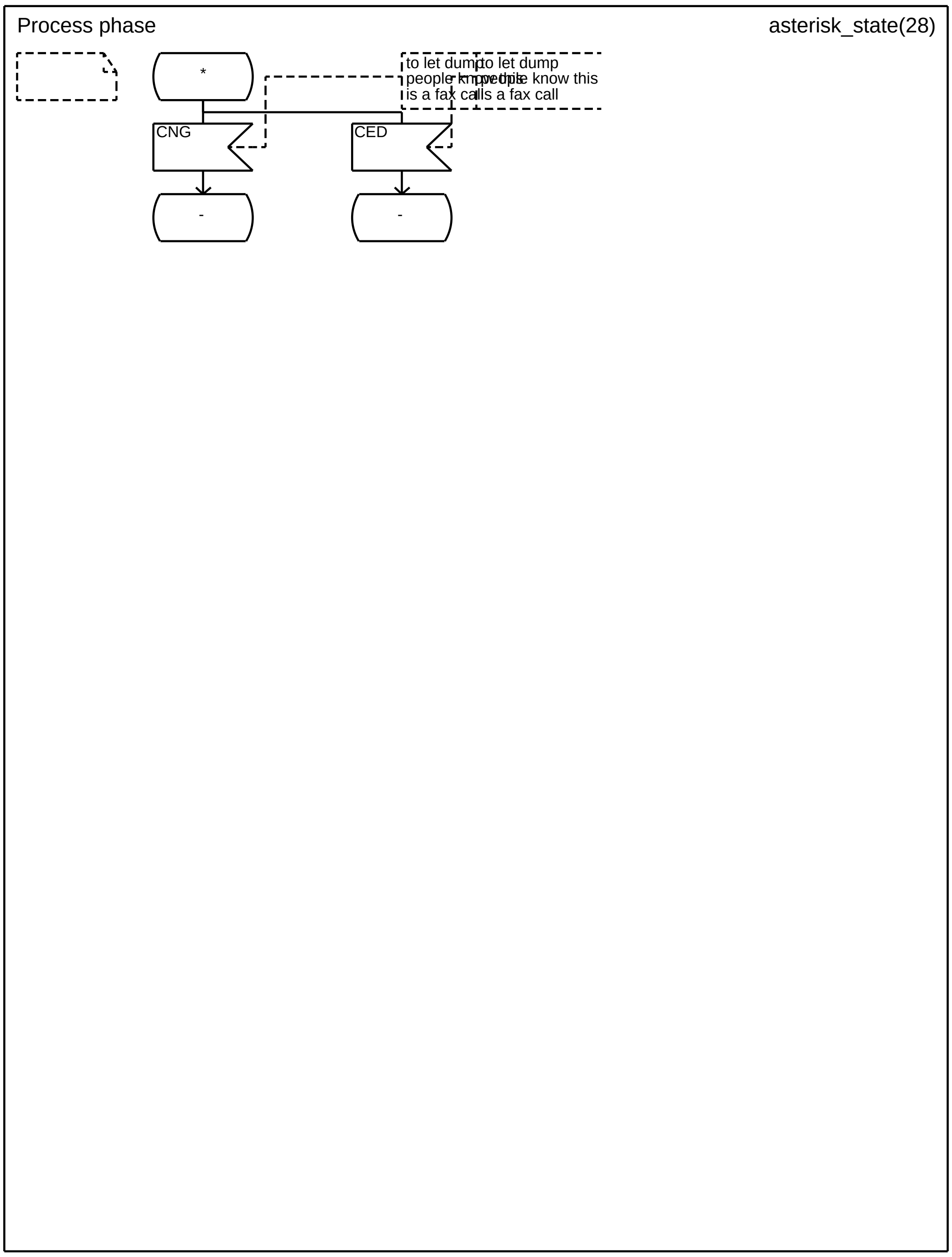

Process phase
PhaseD_MultPage_T1part3(28)
PhaseD_MultPage_T
CRP
(DisData)
'Set mode'
MPS
(DisData) VIA g
RESET(T4Timer)
/* //ETOFLI put this elsewhere */
Wait_CFR

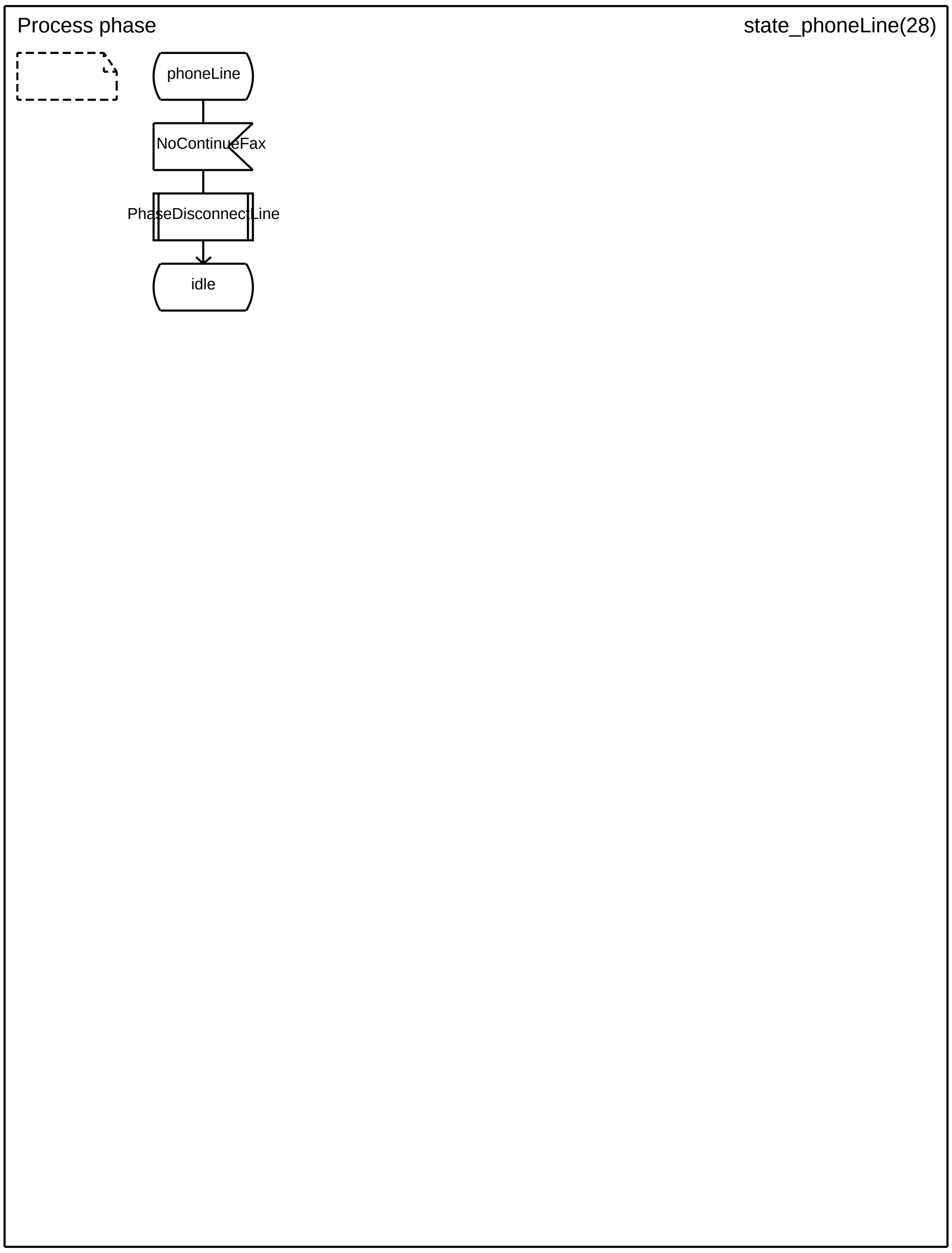

Process phase
AllState(28)
*
PIN (DISData)
RESET(T4Timer)
RESET(T1Timer)
RESET(T2Timer)
//etofli how to set timers again ?
PhaseAltSpeech (RetVal)
RetVal
1
0
2
PhaseDisconnectLine
-
idle
PhaseB_R

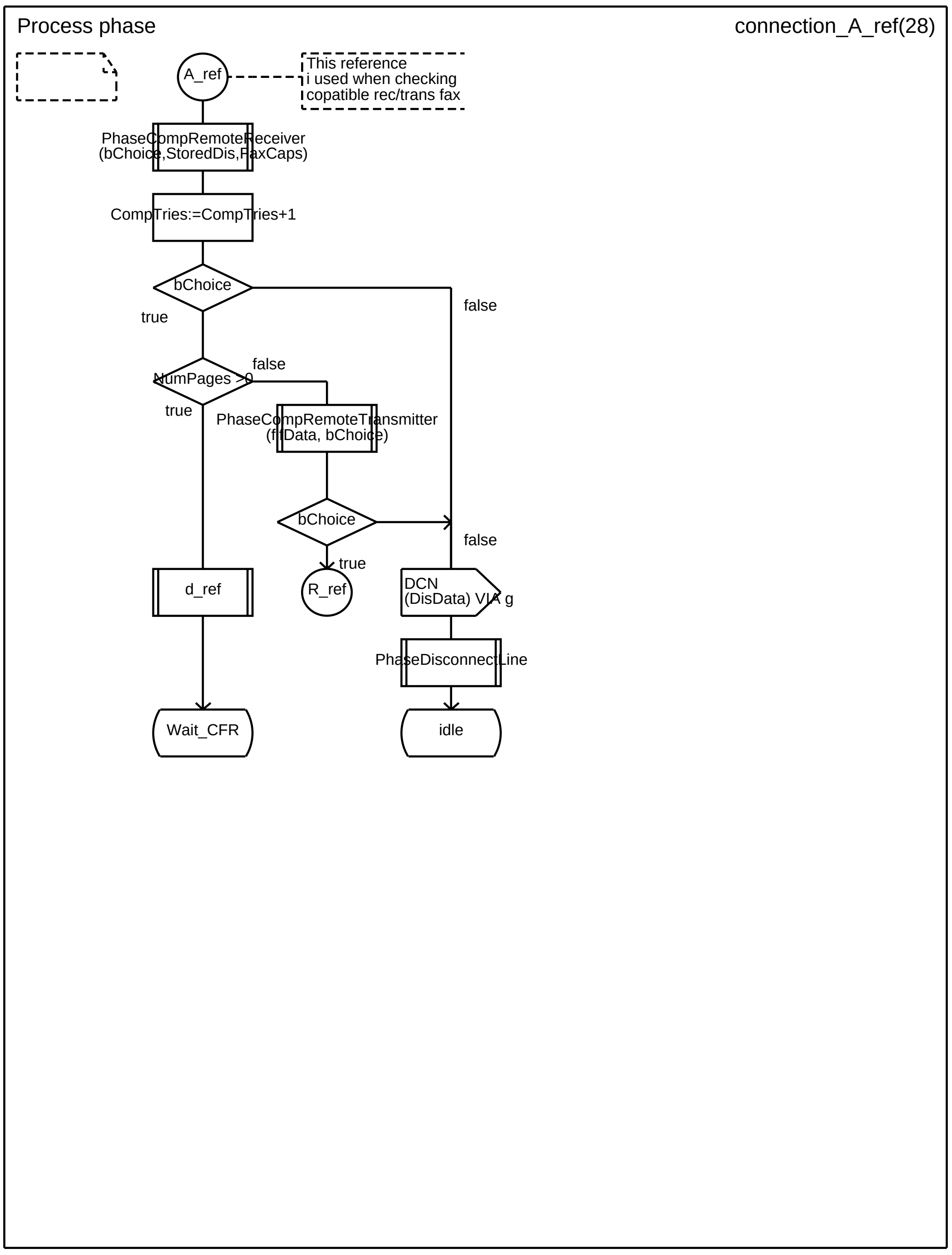

Process phase
AllStatepart2(28)
PhaseD_MultPage_T
PIP
(DISData)
RESET(T4Timer)
RESET(T1Timer)
RESET(T2Timer)
//etofli how to set
timers again ?
PhaseAltSpeech
(RetVal)
RetVal
0
2
1
PhaseDisconnectLine
-
idle
PhaseB_R

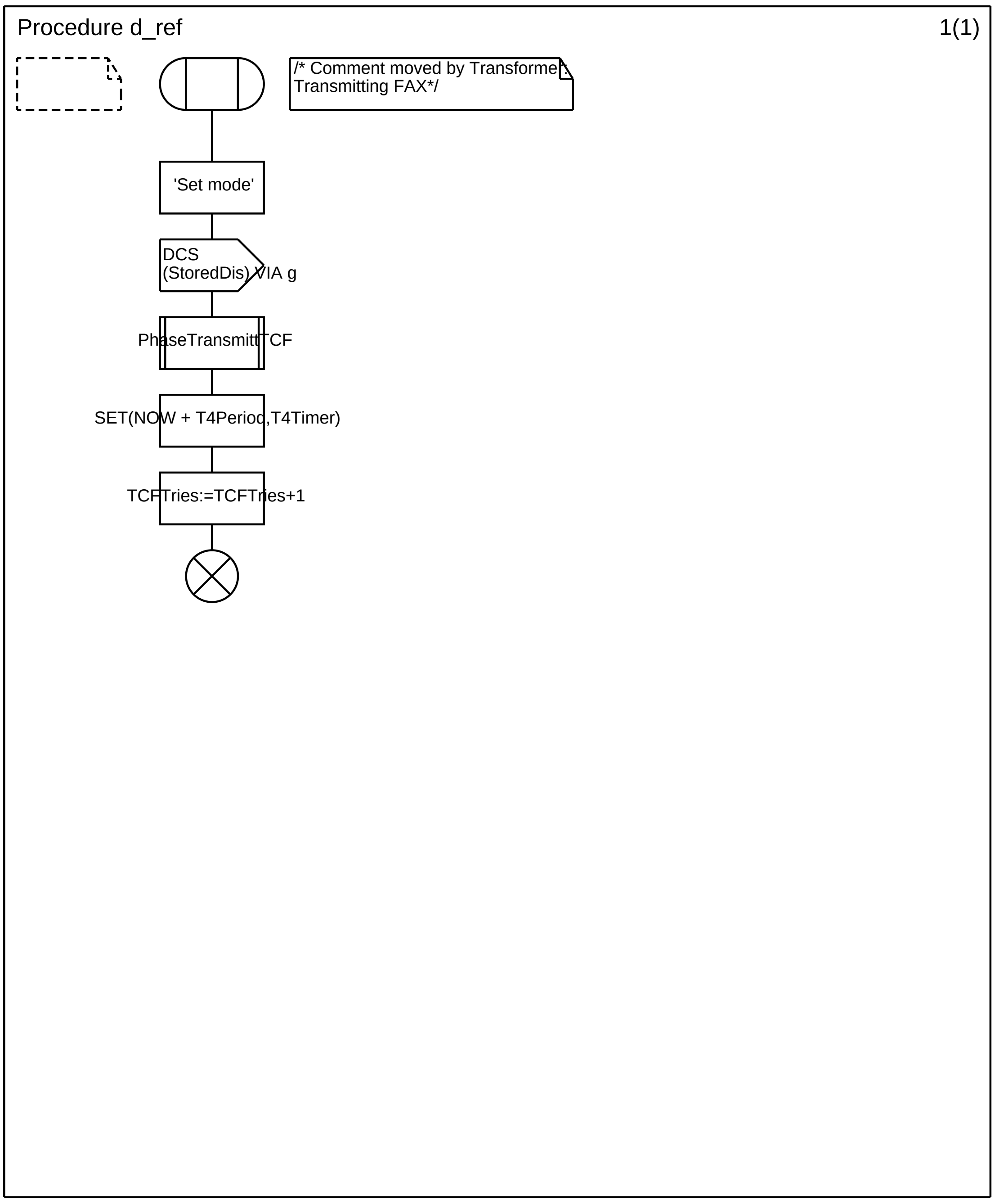

Process phase
PhaseD_T1(28)
phaseD_T
Transmitting FAX in phase D
MCF (DisData)
RTP (DisData)
RTN (DisData)
RESET(T4Timer)
RESET(T4Timer)
RESET(T4Timer)
OkFax via a
NokFax via a
CapableToReXmit = true
false
true
DCN (DisData) VIA g
DCN (DisData) VIA g
d_ref
PhaseDisconnectLine
PhaseDisconnectLine
NokFax via a
DCN (DisData) VIA g
PhaseDisconnectLine
idle
idle
Wait_CFR
idle


Process phase
PhaseD_T1part2(28)
phaseD_T
*
T4Timer
RESET(T4Timer)
Timeouts:=Timeouts+1
NokFax via a
SET(NOW + T4Period,T4Timer)
DCN
(DisData) VIA g
Timeouts=3
true
false
PhaseDisconnectLine
EOP
(DisData) VIA g
NokFax via a
DCN
(DisData) VIA g
PhaseDisconnectLine
idle
phaseD_T
idle


Process phase
asterisk_state(28)
*
CNG
CED
to let dump people know this is a fax call
to let dump people know this is a fax call
-
-


Process phase
state_phoneLine(28)
phoneLine
NoContinueFax
PhaseDisconnectLine
idle


Process phase
connection_A_ref(28)
A_ref
This reference
i used when checking
copatible rec/trans fax
PhaseCompRemoteReceiver
(bChoice,StoredDis,FaxCaps)
CompTries:=CompTries+1
bChoice
true
false
NumPages >0
false
true
PhaseCompRemoteTransmitter
(fiIData, bChoice)
bChoice
false
true
d_ref
R_ref
DCN
(DisData) VIA g
PhaseDisconnectLine
Wait_CFR
idle


Procedure d_ref
1(1)
/* Comment moved by Transformer:
Transmitting FAX*/
'Set mode'
DCS
(StoredDis) VIA g
PhaseTransmittTCF
SET(NOW + T4Period,T4Timer)
TCFTries:=TCFTries+1

APPENDIX E

SOURCE CODE: TRANSFORMER.PL

This appendix contains the source code to the program developed to implement and test the rules from chapter 4. The source code is available in electronic form at `http://www.dahle.no/thesis`.

```
#! /local/bin/perl -s
# Transformer.pl version 0.45, 24.03.99
# Written by Ole Dahle
# Purpose: Rewrite SDL diagrams (in PR/CIF format) to make them more readable
# Usage: perl transfomer.pl [-warn] [-insert] infile [outfile]

#Preferences:
$infile = ”phase3.sdl”;
$showWarnings = 0;
$insertComments = 0;
$removeInvisibleJoins = 1;

$flowsPerPage = 4;  # Number of flows allowed on a page
$flowSpacing = 380; # Number of pixels between each flow
$pageWidth = 1900;
$pageHeight = 2650;

#Look for command line options
if($warn != 0) {$showWarnings = 0;}
if($insert != 0) {$insertComments = 0;}

$firstfile = 1;
foreach (@ARGV)
{ if(!(/^-/) && !$firstfile) {$outfile = $_;}
  if(!(/^-/) && $firstfile) {$infile = $_; $firstfile = 0;}
}
if($outfile eq ””) {$outfile = ”fixed_$infile”;}
if($insertComments)
{$flowsPerPage = $flowsPerPage/2;
$flowSpacing = $flowSpacing *2;}

print ”$infile $outfile $showWarnings $insertComments space: $flowSpacing\n”;

#Global variables
@sdlfile;      #Array that contains the SDL file
$i = $j = 0;   #Counters
$x=  $y = 0;   #Position variables
$statePosX = 0; $statePosY = 0; # Position of state symbol


#Initialize: Open infile, read it into the sdlfile array.
print ”Transformer starting\n”;
open(INPUT,$infile) or die(“Could not open input file $infile.\n”);
print ”opening $infile\n”;
@sdlfile = <INPUT>;
close(INPUT);

#Perform transformations

if($insertComments)
{ &insertTextExtensions;
  &makeSignalTextExtension;
}

if($removeInvisibleJoins)
{ &removeInvisibleJoins;}
&expandConnections;
&stateOnNewPage;
&alignPages;
&resolvePageOverflow;
&alignNextstates;
&removeEmptyPages;

if($showWarnings)
{ &warnShortNames;
  &warnBranchOnDecision;
}



#Write transformed SDL to file

open(OUTPUT,”>$outfile”) or die(“Could not open output file $outfile.\n”);
print ”Writing to $outfile.\n”;
foreach $outline (@sdlfile)
{ print OUTPUT ”$outline”;
}
close(OUTPUT);
```

```
#End of program

#----------- Subroutines representing transformational rules below-----------#

sub expandConnections
{ local(@connections) = ();
  local(@joins) = ();
  local(%definition) = 0;
  local($numberOfConnections) = 0;
  local($invisibleJoins) = 0;
  local(%used) = ();
  local($currentState);


  #Scan trough sdlfile, find all connections;
  $i = 0;
  while($i< @sdlfile)
  { if($sdlfile[$i] =~ /^join (\w+)/i) #Count occurences of join
    { #Lowercase connection name, works with norwegian characters:
      $temp = $1; $temp =~ tr/A-ZÆØÅ/a-zæøå/;

      #Count auto-generated, graphical connections separately
      if($temp =~ /^grst\d+/)
      {$invisibleJoins++;}
      else
      {$used{$temp}++;}
    }
    $i++;
  }

  @temp = %used;
  $numberOf = @temp;
  $numberOf = $numberOf/2;
  print ”There are $numberOf connections and $invisibleJoins invisible joins used in the
 file.\n”;

  # Treat each connection separately
  foreach $con (reverse sort keys(%used))
  { print ”Connection $con was used $used{$con} times. Do you want to:\n”;
    print ”Expand all occurences?      (1)\n”;
    print ”Expand selected occurences? (2)\n”;
    print ”Turn it into a procedure?   (3)\n”;
    print ”Do nothing?     (Any other key)\n”;
    $answer = <>;
    #print ”$answer \n”;

    if($answer ==1 || $answer ==2 || $answer ==1)
    {
    #Rescan sdlfile, since joins-wihtin-joins may have been expanded, and
    # increased the number of joins.
    $i = 0;
    foreach $temp( keys (%used)) {$used{$temp} = 0;}
```

```
     while($i< @sdlfile)
     {if($sdlfile[$i] =~ /^join (\w+)/i) #Count occurences of join
      { #Lowercase connection name, works with norwegian characters:
        $temp = $1; $temp =~ tr/A-ZÆØÅ/a-zæøå/;


        #Skip auto-generated, graphical connections
        if(!($temp =~ /^grst\d+/))
        {$used{$temp}++;}
       }
       $i++;
     }
    }

    # --- Expand connection ---
    if($answer == 1 || $answer == 2)
    { #Finding the definition of the connection
      $i = 0;
      until(($sdlfile[$i] =~ /^connection $con/i) || $i == @sdlfile)
      {$i++};

     if($i == @sdlfile)
      {print ”Couldn’t find the definition of $con!\n”;}
      else
      {
      $def = $i+2;
      $i = 0;
      $expandedJoins = 0;

      #Enter do-it-all or selected-only mode
      if($answer == 1) {$doit = 1;}
      else {$doit = 0; }

      while($i < @sdlfile)
      {  # Keep track of wich state we’re in
         if($sdlfile[$i] =~ /^state (.+)(;|\n)/i) {$currentState = $1;}

        if($answer == 2 &&$sdlfile[$i] =~ /^join $con/i)
         { #Ask if in selected-only mode
           print ”Found an occurence of $con in state $currentState.\n”;
           print ”Expand it? (y/n) ”;
           $expand = <>;
           if($expand =~ /^y/i) {$doit = 1}
         }

         if($sdlfile[$i] =~ /^join $con/i && $doit)
 { $sdlfile[$i-1] =~ /^\/\* CIF Join \((\d+),(\d+)\)/i;
           #print ”$1 $2 $sdlfile[$i-1]”;
           $x=$1; $y=$2;
print ”Replacing an occurence of join $con in state $currentState.\n”;
           splice(@sdlfile,$i-1,2);
```

```
if($def > $i) {$def -= 2;}
        #print ”Definition at $def, pointer at $i\n”;

        $k = $def; $i--;
        #Find the label, find the coordinates
        $tmp = $k;
        until($sdlfile[$tmp] =~ /\/\* CIF/i && $sdlfile[$tmp] !~
        /(Line|comment)/i)
       { # Lines and comments are not at the same coordinates as the flow
          $tmp++;
        }
        $sdlfile[$tmp] =~ /\((\d+),(\d+)\)/;
        #print ”$1,$2 $sdlfile[$tmp]”;
        # Calculating offset:
        $x = $1+50 - $x; $y = $2 - $y;
        #print ”Offset: x: $x y: $y\n”;

        if($sdlfile[$k] =~ /^\/\* CIF Line/i) {$k++;} # Skip first line

        until($sdlfile[$k] =~ /^\/\* CIF End Label/i)
        {
         # Treat comments to the connection label specially:
         if(($sdlfile[$k-1] =~ /^connection/i) &&
           ($sdlfile[$k] =~ /\/\* CIF Comment \((\d+),(\d+)/i))
         { #print ”Moving comment...\n”;
           $tempx = $1-$x; $tempy = $2-$y;

           # Create Textbox instead of comment
           splice(@sdlfile,$i,0,”/* CIF Text ($tempx,$tempy),(300,400) */\n”);
           $i++; $k++;  if($def > $i) {$def++; $k++;}
           splice(@sdlfile,$i,0,”/* Comment moved by Transformer:\n”);
           $i++;  if($def > $i) {$def++; $k++;}

           # Skip the line to the old comment
           if($sdlfile[$k] =~ /\/\* CIF Line/i) {$k++;}

           # Remove ”comment ’” from first line
           splice(@sdlfile,$i,0,”$sdlfile[$k]”);
           print ”Should be old comment: $sdlfile[$i]”;
           $sdlfile[$i] =~ s/comment ’//i;
           $i++; $k++; if($def > $i) {$def++; $k++;}


           # Put the rest of the comment in the box
           until($sdlfile[$k] =~ /^(;|:)/)
           # ’:’ is wrong, but used sometimes?
           { splice(@sdlfile,$i,0,”$sdlfile[$k]”);
           #$sdlfile[$i] =~ s/\n/\*\/\n/;
           $i++; $k++; if($def > $i) {$def++; $k++;}
           }

           # Insert text box end instead of ’;’
           print ”$sdlfile[$k]”;
          $sdlfile[$i-1] =~ s/’/\*\//; print ”$sdlfile[$i-1]”;
           splice(@sdlfile,$i,0,”/* CIF End Text */\n”);
           $i++; $k++; if($def > $i) {$def++; $k++;}

         }
         else
       {splice(@sdlfile,$i,0,$sdlfile[$k]);

         # In some CIF statements, only the first (x,y)
         # pair must be changed,
         # since the second (x,y) defines the size of the symbol.
         if($sdlfile[$i] =~ /\/\* CIF (Label|Join|Comment|Text) /i)
         { $sdlfile[$i]=~ s/\((\d+)/$1-$x/e;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$i]=~ s/(\d+)\)/$1-$y/e;
           # Substitute ’y)’ with y minus offset
           $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2\)/;
           # Put the parantheses back in
         }

         elsif($sdlfile[$i] =~ /\/\* CIF/)
         { $sdlfile[$i]=~ s/\((\d+)/$1-$x/ge;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$i]=~ s/(\d+)\)/$1-$y/ge;
           # Substitute ’y)’ with y minus offset
           $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2\)/g;
           # Put the parantheses back in
         }
          #print ”$sdlfile[$i]”;
          $i++; $k++; if($def > $i) {$def++; $k++;}
        } #End comment /normal line
        } #End inserting
        if($answer ==2) {$doit = 0;}
           #Don’t expand next one without asking
        $expandedJoins++;
      } #End expanding occurence
   $i++;
   }


   #Deleting Connection definition
   if($expandedJoins == $used{$con})
   #Only delete if all occurences were expanded
   {
   print ”Deleting  definition of $con\n”;
   $prevline = ””;
   $i = 0;
   until($sdlfile[$i] =~ /^connection $con/i)
   {$i++};
   $i--;
   # Start with the CIF label-statement above the connection statement
```

```
      until($prevline =~ /^endconnection/i)
       { $prevline = $sdlfile[$i]; #print ”$prevline”;
         splice(@sdlfile,$i,1);
       }
      } #End deleting definition

    } #End found definiton

    } #End expanding connection


  # --- Turn connection into procedure
  if($answer == 3)
    { #Finding the definition of the connection
      $i = 0;
      until(($sdlfile[$i] =~ /^connection $con/i) || $i == @sdlfile)
{$i++};

     if($i == @sdlfile)
      {print ”Couldn’t find the definition of $con!\n”;}
      else
      {
      $def = $i+1;

     until(($sdlfile[$i] =~ /^endconnection $con/i) || $i == @sdlfile)
{ if($sdlfile[$i] =~ /^nextstate (\w+)/i)
    {$returns++; $returnState = $1;}
         $i++;
        }

    #@temp = %returns; $temp = @temp;
    if($returns > 2)
      { print ”Connection $con has returns to more than one state!\n”; }
    else
      { #Open a file to put the new procedure in
        print ”Creating procedure $con in separate file...\n”;
        $filename = ”$con.sdl”;
        $overwrite = ”n”;
        until(!(-e ”$filename”) || $overwrite =~ /^y/i)
        { print ”$filename exists. OK to overwrite? (y/n)\n”;
          $overwrite = <>;
          if($overwrite =~ /^n/i)
          { print ”Please supply a different name: ”;
            $filename=<>;
            chop($filename);
          }
        }

        #Print the procedure to the file
        open(PROCFILE, ”>$filename”) or die(“Failed to open $filename.\n”);

        print PROCFILE ”/* CIF ProcedureDiagram */\n”;
```

```
 print PROCFILE ”/* CIF Page 1 (1900,2300) */\n”;
 print PROCFILE ”/* CIF Frame (0,0),(1900,2300) */\n”;
 print PROCFILE ”/* CIF Specific SDT Version 1.0 */\n”;
 print PROCFILE
”/* CIF Specific SDT Page 1 Scale 100 Grid (250,150) AutoNumbered */\n”;
 print PROCFILE ”Procedure $con;\n”;
 print PROCFILE ”/* CIF DefaultSize (200,100) */\n”;
 print PROCFILE ”/* CIF CurrentPage 1 */\n”;
 print PROCFILE ”/* CIF ProcedureStart (300,100) */\n”;
 print PROCFILE ”start ;\n”;

# Extract coordinates from label  symbol, evaluate offset
$j = $def;
$sdlfile[$j-2] =~ /\/\* CIF Label \((\d+),(\d+)\)/i;
#print ”$sdlfile[$j-2]”;
$x = $1-350;
$y = $2-100;
#print ”x: $x y: $y $sdlfile[$j-2]\n”;

# Print the connection to the file,
# alter all coordinates according to offset
until($sdlfile[$j] =~ /^\/\* CIF End Label/i)
{
    # Treat comments to the connection label specially:
     if(($sdlfile[$j-1] =~ /^connection/i) &&
       ($sdlfile[$j] =~ /\/\* CIF Comment \((\d+),(\d+)/i))
    { print ”Moving comment...\n”;
      $tempx = $1-$x; $tempy = $2-$y;

      # Create Textbox instead of comment
     print PROCFILE ”/* CIF Text ($tempx,$tempy),(300,400) */\n”; $j++;
      print PROCFILE ”/* Comment moved by Transformer:\n”;


       # Skip the line to the old comment
       if($sdlfile[$j] =~ /\/\* CIF Line/i) {$j++;}

       # Remove ”comment ’” from first line
         print ”Should be old comment: $sdlfile[$i]”;
       $sdlfile[$j] =~ s/comment ’//i;
        $sdlfile[$j] =~ s/’\n/\*\/\n/; # If it’s also the last
        print PROCFILE ”$sdlfile[$j]”; $j++;


       # Put the rest of the comment in the box
       until($sdlfile[$j] =~ /^(;|:)/)
       # ’:’ is wrong, but used sometimes?
       {  $sdlfile[$j] =~ s/’\n/\*\/\n/;
           print PROCFILE ”$sdlfile[$j]”;
         $j++;
       }
```

```
              # Insert text box end instead of ’;’
              print ”$sdlfile[$j]”;
              print PROCFILE ”/* CIF End Text */\n”;
              $j++;

            }

         # In a CIF label or Join statement, only the first (x,y)
         # pair must be changed,
         # since the second (x,y) defines the size of the symbol.
         if($sdlfile[$j] =~ /\/\* CIF (Label|Join|Comment|Text) /i)
         { $sdlfile[$j]=~ s/\((\d+)/$1-$x/e;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$j]=~ s/(\d+)\)/$1-$y/e;
           # Substitute ’y)’ with y minus offset
           $sdlfile[$j]=~ s/(\d+),(\d+)/\($1,$2\)/;
           # Put the parantheses back in
         }

         elsif($sdlfile[$j] =~ /\/\* CIF/)
         { $sdlfile[$j]=~ s/\((\d+)/$1-$x/ge;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$j]=~ s/(\d+)\)/$1-$y/ge;
           # Substitute ’y)’ with y minus offset
           $sdlfile[$j]=~ s/(\d+),(\d+)/\($1,$2\)/g;
           # Put the parantheses back in
         }

         #Turn the nextStates into returns
         if($sdlfile[$j] =~ /\/\* CIF NextState \((\d+),(\d+)/i)
         { $x = $1+50; $y = $2;
           $sdlfile[$j] = ”/* CIF Return($x,$y),(100,100) */\n”;
           $sdlfile[$j+1] = ”return;\n”;
         }
         print PROCFILE ”$sdlfile[$j]”;
         $j++;
       }

       #Print the last part of the procedure to the file.
       print PROCFILE ”/* CIF End ProcedureDiagram */\n”;
       print PROCFILE ”endprocedure $con;\n”;

       close(PROCFILE);

       #Insert reference to the procedure in the beginning of sdlfile
       $i=0;
       until($sdlfile[$i] =~ /^\/\* CIF CurrentPage/i) {$i++;}
       splice(@sdlfile,$i+1,0,”/* CIF Procedure (600,100) */\n”);
       splice(@sdlfile,$i+2,0,”/* CIF TextPosition (625,125) */\n”);
       splice(@sdlfile,$i+3,0,”procedure $con referenced;\n”);

       #Search through sdlfile and replace all joins
       #to the connection with procedure calls
       $i = 0;
       while($i<@sdlfile)
       {
       if($sdlfile[$i] =~ /^join $con/i)
         { $sdlfile[$i-1] =~ /^\/\* CIF Join \((\d+),(\d+)\)/i;
           $x=$1-50; $y=$2;
           #print "Replacing an occurence of $sdlfile[$i]\n";
           $sdlfile[$i-1] = "/* CIF ProcedureCall ($x,$y) */\n";
           $sdlfile[$i] = "call $con;\n";
           $x = $x+100; $y = $y+100; $y2 = $y+50;
           splice(@sdlfile,$i+1,0,"/* CIF Line ($x,$y),($x,$y2) */\n");
           $x = $x-100;
           splice(@sdlfile,$i+2,0,"/* CIF NextState ($x,$y2) */\n");
           splice(@sdlfile,$i+3,0,"nextstate $returnState;\n");
         } #End replacing join
        $i++;
       } #End replacing all joins with calls

      #Deleting Connection definition
      print "Deleting  definition of $con\n";
      $prevline = "";
      $i = 0;
      until($sdlfile[$i] =~ /^connection $con/i)
      {$i++};
      $i--;
      # Start with the Label CIF statement above the connection statement
      until($prevline =~ /^endconnection/i)
       { $prevline = $sdlfile[$i]; #print "$prevline";
         splice(@sdlfile,$i,1);
       }

      } #End only one return
      } #End found definition
    } #End turn into procedure

  } # End Treating connections

}

sub stateOnNewPage
{ #Purpose: Insert pagebreaks before states and connection,
  #see rule 'state and nextstate'(1)
  local($line) = "";
  local($statename) = "";
  local($notfound) = 1;
  local($pagedeclare) = 0;

  $lenght = @sdlfile;
  $i = 0;

  # Find the place in the CIF file header where the pages are declared
  while(($sdlfile[$i] =~ /(CIF ProcessDiagram)|(CIF Page)|(CIF Frame)/i))
  {$i++;
  }
  $pagedeclare = $i-1;


  # Search trough sdlfile, looking for states and connections
  while($i< @sdlfile)
  {
    if( $sdlfile[$i] =~ /^(state|connection) (\w+|\*)/i)
    {
      $statename = ”$1_$2”;
      if($statename eq ”state_*”)
      { print ”Changed name on _*\n”;
        $statename = ”asterisk_state”;
      }
      # print ”checking $statename\n”;

      # Search backwards in sdlfile, to see if a pagebreak
      # has been declared since
      # last endstate or endconnection
      $j = $i-1; $notfound = 1;
      while(!($sdlfile[$j] =~ /^(endstate|endconnection)/) && $notfound)
      { if($sdlfile[$j] =~ /^\/\* CIF CurrentPage/)
        { $notfound = 0; #print ”$statename HAS a pagebreak\n”;
        }
        else {$j--; #print ”.”;
             }
      }

      if($notfound) # Insert code for pagebreak
      { splice(@sdlfile,$j+1,0,”/* CIF CurrentPage $statename */\n”);
        $i++; $lenght++;

        # Insert declaration of the new page in the CIF header
        splice(@sdlfile,$pagedeclare+1,0,
        ”/* CIF Page $statename ($pageWidth,$pageHeight) */\n”);
        splice(@sdlfile,$pagedeclare+2,0,
        ”/* CIF Frame (0,0),($pageWidth,$pageHeight) */\n”);
        $pagedeclare = $pagedeclare+2;
        $i=$i+2;
        #Skip two lines, since the declaration has moved everything downwards

         print ”Inserted new page for $statename\n”;
      } # End insert pagebreak

    } # End backtracking

    $i++;
  } # End searching through sdlfile

}
```

```
sub alignPages
{ # Prupose: To make state symbols appear in the upper left corner
  local($pageStart) = 0;

  $i = 0;
  print ”aligning pages\n”;

  # Search through sdlfile, check all pages
  while($i < @sdlfile)
  { if(($sdlfile[$i] =~ /\/\* CIF CurrentPage/i)
    && ($sdlfile[$i+1] =~ /\/\* CIF (State|Label)/i))
                                   # Give up on the start page :-(
    # Page with state found
   { $i++;
     $sdlfile[$i] =~ /\/\* CIF (State|Label|Start) \((\d+),(\d+)\)/i;
     # Put the state symbol in the right place.
     # This might brake the lines to the symbol, this will be fixed by
     # resolvePageOverflow. Oops, there goes procedure independency...
     if($1 =~ /label/i) {$x = 350;} else {$x = 300;}
     if(($2 != $x) || ($3 != 100))
       { $sdlfile[$i]=~ s/\((\d+)/$x/e; # Substitute ’(x’ with the right x
         $sdlfile[$i]=~ s/(\d+)\)/100/; # Substitute ’y)’ with 100
         $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2\)/; #Put the parantheses back in
       }
    $pageStart = $i;
    $i++;
    #print ”$sdlfile[$i]\n”;
    # Extract coordinates for the first symbol, evaluate offset
    until($sdlfile[$i] =~ /^\/\* CIF/i && ($sdlfile[$i] !~ /(Line|Comment)/i))
      {$i++;}
    $sdlfile[$i] =~ /^\/\* CIF \w+ \((\d+),(\d+)\)/i;
    $x = $1-300;
    $y = $2-250;
    print ”line: $i x: $x y: $y $sdlfile[$i]”;

    # Search through the page, alter all coordinates according to offset
    $i = $pageStart+1; # Jump back up to the line after state statement
    until($sdlfile[$i] =~ /^(endstate|endconnection)/i)
    {
      # In some CIF statements, only the first (x,y) pair must be changed,
      # since the second (x,y) defines the size of the symbol.
      if($sdlfile[$i] =~ /\/\* CIF (Label|Join|Comment|Text)/i)
      {
        $sdlfile[$i]=~ s/\((\d+)/$1-$x/e;
        # Substitute ’(x’ with x minus offset
        $sdlfile[$i]=~ s/(\d+)\)/$1-$y/e;
        # Substitute ’y)’ with y minus offset
        $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2\)/;
        # Put the parantheses back in
        #print ”Replaced only 1st x,y on line $i: $sdlfile[$i]”;
      }

      elsif($sdlfile[$i] =~ /\/\* CIF/)
      { $sdlfile[$i]=~ s/\((\d+)/$1-$x/ge;
        # Substitute ’(x’ with x minus offset
        $sdlfile[$i]=~ s/(\d+)\)/$1-$y/ge;
        # Substitute ’y)’ with y minus offset
        $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2\)/g;
        # Put the parantheses back in
      }
      $i++;
    } # End aligning symbols

    # Adjust nextstates upwards if they are far below the previous symbol
    $i = $pageStart;

    until($sdlfile[$i] =~ /^(endstate|endconnection)/i)
    {
     if($sdlfile[$i] =~ /^\/\* CIF NextState/i)
     {  #Search upwards to find the line to the nextstate
        $rightline = $i-1;
        until($sdlfile[$rightline] =~ /\/\* CIF Line/i)
        {$rightline--; }

     $sdlfile[$rightline] =~ /\/\* CIF Line \(\d+,(\d+)\),\(\d+,(\d+)\) \*\//;
       #Extract the two y-coordinates for the line
       #print ”$sdlfile[$rightline]”;
       #print ”y1: $1 y2: $2\n”;
       if($2-$1 != 50)
       { $newY = $1+50;
         $sdlfile[$rightline]=~ s/(\d+)\) \*\//$newY\) \*\//;
         #Adjust the second y-coordinate
         $sdlfile[$i]=~ s/(\d+)\) \*\//$newY\) \*\//;
       }#End moving misplaced nextstate
     } #End checking nextstate
     $i++;
    } #End checking page for misplaced nextstates


   } # End found page
    $i++;
  } # End searching through sdlfile
}


sub alignNextstates
{ #Purpose: Align nextstates on each page horizontally,
  #see rule ’state and nextstate’(2)

  local(@statelines) = ();
  local($nextstates) = 0;
  local($largestY) = 0;
  local($line) = ””;
  local($currentState) = ”start”;

  $i = 0;
  $lenght = @sdlfile;

print ”Aligning nextstates\n”;


while($i < @sdlfile)
{ @statelines = ();
  $nextstates = 0;


  # For each page put the linenumbers of nextstate symbols in @statelines,
  until($sdlfile[$i] =~ /^(\/\* CIF CurrentPage|endprocess|endprocedure)/i)
  { if(($sdlfile[$i] =~ /^nextstate/) && ($currentState ne ”start”))
    {                    # Don’t move the nextstate in the start transition
      #print”$sdlfile[$i-1]”;
      #print”$sdlfile[$i]\n”;
      $statelines[$nextstates] = $i;
      $nextstates++;
    }

    # Keep track of wich state we’re in
    if($sdlfile[$i] =~ /^state (.+);/i)
    {$currentState = $1;
    }

    $i++;
  }

  # Compare the y-coordinates and put the largest in $largestY
  $largestY = 0;

  foreach $line (@statelines)
  { #print ”find Y: $sdlfile[$line-1]\n”;
    $sdlfile[$line-1] =~ /(\d+)\) \*\//;
    if($1 > $largestY) {$largestY = $1;}
  }
#  print ”y: $largestY\n”;

   # Substitute the y-coordinate for the nextstate symbols and the line
   # to them with $largestY
   foreach $line (@statelines)
   { $sdlfile[$line-1] =~ s/(\d+)\) \*\/$/$largestY\) \*\//;

     #Search upwards to find the line to the nextstate
     $rightline = $line-2;
     until($sdlfile[$rightline] =~ /\/\* CIF Line/i)
     {$rightline--; }
     $sdlfile[$rightline] =~ s/(\d+)\) \*\/$/$largestY\) \*\//;
     #print"$sdlfile[$rightline]";
     #print"$sdlfile[$line-1]\n";
    }
  $i++; #Go past the pagebreak and into the next page
} #end searching through sdlfile
}

sub makeSignalTextExtension
{ # Purpose: Put the parameters to a signal in a text extension if necessary,
  # see rule 'signal parameters'
   $i = 0;
   local($parameters) = 0;
   local($signalLine) = 0;
   local($CIFstring) = "";
   local($extFound) = 0;
   local($signalname) = "";

  print "Checking signals for necessary text extentions\n";
  while($i < @sdlfile)
  {
    # If the signal has parameters, start checking it
    if($sdlfile[$i] =~ /^(output|input) (\w+)/i && !($sdlfile[$i] =~ /;/))
    { $signalname = $2;
      $parameters = 0; $extFound = 0;
      $signalLine = $i;

      # Count the parameters, every line beginning with a letter,
      # except 'comment'
      until(($sdlfile[$i] =~ /(;|^\/\* CIF Comment)/i) || ($extFound == 1))
      { if($sdlfile[$i] =~ /^\(?\w+/ && !($sdlfile[$i] =~ /^comment/))
        { $parameters++;}

        if($sdlfile[$i] =~ /CIF TextExtension/i)
        { $extFound = 1; print "$signalname HAS text extention\n";
        }
      $i++;
      }

      if($parameters > 2 && ($extFound == 0))
      { print "Added extension to signal $signalname\n";
        # Extract the coordinates of the signal symbol
        $sdlfile[$signalLine-1] =~ /\/\* CIF .+put \((\d+),(\d+)\)/;
        $x = $1; $y = $2;

        # Insert CIF code for a text extension and
        # line rigt after the signal statement
        $CIFstring = sprintf("/* CIF TextExtension (%d,%d)
        Right */\n",$x+250,$y);

        splice(@sdlfile,$signalLine+1,0,$CIFstring);
        $CIFstring = sprintf("/* CIF Line (%d,%d),
        (%d,%d) */\n",$x+250,$y+50,$x+200,$y+50);
        splice(@sdlfile,$signalLine+2,0,$CIFstring);

# Insert CIF code to end the text extension right before the semicolon
        splice(@sdlfile,$i+2,0,"/* CIF End TextExtension */\n");
                      # $i+2, since two lines have been inserted
        $i=$i+3;
      } # End make text extension

    } # End check signal statement
    $i++;
  } # End search through sdlfile

}

sub removeInvisibleJoins
{ # Purpose: Expand all invisible joins, remove the invisble labels.
  # See rule 'connections'

  local($joinName) = "";
  local($notFound) = 1;

 # Search trough sdlfile, expand all invisible joins
 print "Expanding invisible joins.\n";
 $i = 0;
 while($i< @sdlfile)
 { if($sdlfile[$i] =~ /^\/\* CIF Join Invisible/i)
   { $sdlfile[$i+1] =~ /join (\w+);/i;
     $joinName = $1;
     #print "Found invisible join $joinName at line ",$i+1,".\n";

     # Find the coordinates of the join
     if($sdlfile[$i-1] =~ /^\/\* CIF Line \((\d+),(\d+)/)
       {$joinLine = $i-1;  $x = $1-100; $y =$2+50;
        print "x: $x y:$y $sdlfile[$joinLine]";
       }
     elsif($sdlfile[$i+2] =~ /^\/\* CIF Line \((\d+),(\d+)/)
       {$joinLine = $i+2; $x = $1-100; $y =$2+50;
        print "x: $x y:$y $sdlfile[$joinLine]";
       }
     else {print "Couldn't find coordinates!\n"; }

     # Search for the definition of the connection
     $j= 0; $notFound = 1;
     while(($j < @sdlfile) && $notFound)
     { if(($sdlfile[$j] =~ /^$joinName/i) &&
         ($sdlfile[$j-1] =~ /^\/\* CIF Label Invisible/i))
         {$notFound = 0; print "Found $sdlfile[$j]";}
       else {$j++;}
     } # End finding definition
     #print "Definition at line $j: $sdlfile[$j]";
     # Remove the join
     #print ”deleting: $sdlfile[$i]”;
     splice(@sdlfile,$i,1);
     #print ”deleting: $sdlfile[$i]”;
     splice(@sdlfile,$i,1);
     # Make the line to the expanded symbols
     $orgY = $y-50; $tmp = $x+100;
     $sdlfile[$joinLine] = ”/* CIF Line ($tmp,$orgY),($tmp,$y) */\n”;

     # Find the coordinates of the first symbol after
     # the invisible join, evaluate offset
     $sdlfile[$j+1] =~ /\((\d+),(\d+)\) \*\/$/;
     $x = $1 - $x; $y = $2 - $y; print ”offset: $x, $y\n”;

     # Expand the connection
     $j++; print ”Expanding $joinName:\n”;
     until($sdlfile[$j+1] =~ /^\/\* CIF Input/i ||
           $sdlfile[$j-1] =~ /^(endstate|endconnection)/i)
     { splice(@sdlfile,$i,0,$sdlfile[$j]);
       if($j>$i) {$j++;}

       #Correct the coordiantes in the inserted line
       # In a CIF label or Join statement,
       # only the first (x,y) pair must be changed,
       # since the second (x,y) defines the size of the symbol.
         if($sdlfile[$i] =~ /\/\* CIF (Label|Join|Comment|Text) /i)
         { $sdlfile[$i]=~ s/\((\d+)/$1-$x/e;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$i]=~ s/(\d+)\)/$1-$y/e;
           # Substitute ’y)’ with y minus offset
           $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2\)/;
           # Put the parantheses back in
         }

         elsif($sdlfile[$i] =~ /\/\* CIF/)
         { $sdlfile[$i]=~ s/\((\d+)/$1-$x/ge;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$i]=~ s/(\d+)\)/$1-$y/ge;
           # Substitute ’y)’ with y minus offset
           $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2\)/g;
           # Put the parantheses back in
         }
       #print ”$sdlfile[$i]”;
        $i++; $j++;
       } # End expanding connection

   } # End found invisible join
 $i++;
 } # End expanding invisible joins

 #Search trough sdlfile, remove all invisible labels
```

```
 $i = 0;
 while($i< @sdlfile)
 { if ($sdlfile[$i] =~ /^\/\* CIF Label Invisible/i)
   {  print ”Removing label $sdlfile[$i+1]”;
      splice(@sdlfile,$i,1);
      splice(@sdlfile,$i,1);
   }

 $i++;
 } # End deleting labels

} # End sub


sub insertTextExtensions
{ # Purpose: Warn of missing TO/FROM statements/comments,
  # insert dummy comments in the SDL,
  # see rule ’source and destination’


  $i = 0;
  local($found) = 0;
  local($commentString) = ””;
  local($signalname) = ””;
  local($direction) = ””;

  print ”Checking TO/FROM-statements on signals\n”;
  while($i< @sdlfile)
  { $sdlfile[$i]; $i++;

    # If a signal statement is found, extract the signal name and coordinates
    if($sdlfile[$i] =~ /^\/\* CIF (in|out)put \((\d+),(\d+)\)/i)
    { $direction = $1; $x = $2; $y = $3;
      #print ”dir: $direction\n”;
      $sdlfile[$i+1] =~ /put (\w+)/;
      $signalname = $1;
      $found = 0;

      # Check if the signal has a TO/FROM-statement or comment
      until(($sdlfile[$i] =~ /;$/) || $found)
      { $i++;
        if($sdlfile[$i] =~ /^(comment ’(TO|FROM)|TO|VIA)/i)
        {$found = 1;}
      }

      # If the signal has no TO/FROM statement/comment, insert a comment
      if(!($found))
      {
         if($sdlfile[$i] =~ /^$directionput/)
         # If the semicolon is on the same line as the signal name

  { $sdlfile[$i] =~ s/;//;
```

```
            $i++;
            splice(@sdlfile,$i,0,”;\n”); # Put the semicolon on the next line
          }

        if($direction =~ /in/i)
        { $direction = ”FROM”;}
        else {$direction = ”TO”;}

        # Insert the CIF code for the comment symbol, a line and the text
     $commentString =
     sprintf(“/* CIF Comment (%d,%d) Right */\n”,$x+250,$y);
     splice(@sdlfile,$i,0,$commentString);
     $commentString =
     sprintf(“/* CIF Line (%d,%d), (%d,%d) Dashed */\n”,$x+250,$y,$x+200,$y);
        splice(@sdlfile,$i+1,0,$commentString);
        $commentString = ”comment ’$direction: ?’\n”;
        splice(@sdlfile,$i+2,0,$commentString);

        print ”Warning: signal $signalname has no $direction-statement\n”;
        #print ”Added dummy $direction-statement to $signalname\n”;

      } # End insert comment
    } # End searching through signal statement
  } # End searching through sdlfile
}

sub warnBranchOnDecision
{ # Purpose: Warn on decisions in the SDL, see rule ’control flow’


  $i = 0;
  local($currentState);

  while($i< @sdlfile)
  { $line = $sdlfile[$i]; $i++;

    # Keep track of wich state we’re in
    if($line =~ /^state (.+);/i)
    {$currentState = $1;
    }

    if($line =~ /decision (.+);/i)
    { print ”Warning: in state $currentState,
             the process branches on decision: $1\n”;
    }
  }
}

sub warnShortNames
{  # Purpose: Warn on short names in the SDL, see rule ’meaningful names’
```

```
  $i = 0;

  while($i< @sdlfile)
  { $line = $sdlfile[$i]; $i++;
    if($line =~ /(DCL|SIGNAL) (\w+)/i)
    {  $name=$2; $len = length($2);
       if($len < 5)
{ if($1 eq ”DCL”) { print ”Warning: variable $name has a short name\n”;}
          if($1 eq ”SIGNAL”)
          { print ”Warning: signal $name has a short name\n”;}
        }
    } # End check declaration
  } # End search through sdlfile
}

sub resolvePageOverflow
{ # Purpose: Make sure there is enogh horizontal space on each page, if not
  # create new pages.

  local($stateName) = ””;
  local($pageName) = ””;
  local($rightX) = 0;
  local($newPages) = 0;
  local($flowsUsed) = 0;
  local($startTransition) = 0;
  local($spaceNeeded) = 0;
  local($isState) = 1;
  local($lastName) = ””;

  print ”Checking horizontal spacing.\n”;
  $i = 0;

while($i< @sdlfile)
{ if($sdlfile[$i] =~ /^\/\* CIF CurrentPage (\w+)/i)
  { $pageName = $1; $lastName = $pageName;

    until($sdlfile[$i] =~ /^\/\* CIF (State|Label)/i)
      {$i++;}

   if($sdlfile[$i] =~ /^\/\* CIF (State|Label) \((\d+),(\d+)/i)
      {if($1 eq ”State”){$isState = 1;
          $statePosX = $2+100; $statePosY = $3+100;}
       if($1 eq ”Label”){$isState = 0; $statePosX = $2; $statePosY = $3+100;}
      }

    if($sdlfile[$i+1] =~ /^(state|connection) (\w+)/i)
      {$stateName = $2;}

    $flowsUsed = 0;
    $newPages = 1;
    $i++;
  while($sdlfile[$i] !~ /^\/\* CIF CurrentPage/i && $i < @sdlfile)
  {


    if($sdlfile[$i] =~ /^(nextstate|join)/)
      { # Check that it’s not the start transition
        $j = $i; $startTransition = 0;
        while(($sdlfile[$j] !~ /^(state|connection)/i) && !$startTransition)
        { if($sdlfile[$j] =~ /^start/i)
            {$startTransition = 1; #print ”Is in start Transition?\n”;
          }
          $j--;
        } # End check for start transition
        if(!$startTransition)
          {$flowsUsed++;}
      }

    if($sdlfile[$i] =~ /^(input|save)/i)
    { $spaceNeeded = 0;
      if($sdlfile[$i-1] =~ /^\/\* CIF (Input|Save) \((\d+),(\d+)/)
      {
        $symbolX = $2; $symbolY = $3; print ”$symbolX $symbolY $sdlfile[$i]”;
        if($1 =~ /save/i) {$spaceNeeded++;}
      }
      else {print ”Couldn’t find symbolX in line ”,$i-1,”: $sdlfile[$i-1]”;}

      # Check space needed by the flowline
      $j = $i+1;
      while($sdlfile[$j] !~ /^(input|endstate|endconnection|save)/i)
      { if($sdlfile[$j] =~ /^(nextstate|join)/i) {$spaceNeeded++;}
        $j++;
      }
       print ”Space: $spaceNeeded used: $flowsUsed\n”;
      if($spaceNeeded+$flowsUsed> $flowsPerPage)
      { # Create new page
        $newPages++;  $flowsUsed = 0;
        print ”Created new page $pageName\part$newPages\n”;
        if($isState)
        { splice(@sdlfile,$i-2,0,”/* CIF End State */\n”);
          $i++;
          splice(@sdlfile,$i-2,0,”endstate;\n”);
          $i++;
  splice(@sdlfile,$i-2,0,”/* CIF CurrentPage $pageName\part$newPages */\n”);
          $i++;
          splice(@sdlfile,$i-2,0,”/* CIF State (300,100) */\n”);
          $i++;
          splice(@sdlfile,$i-2,0,”state $stateName;\n”);
          $i++;
       }
        else
{ splice(@sdlfile,$i-2,0,”/* CIF End Label */\n”);
          $i++;
  splice(@sdlfile,$i-2,0,”endconnection;\n”);
          $i++;
  splice(@sdlfile,$i-2,0,”/* CIF CurrentPage $pageName\part$newPages */\n”);
          $i++;
  splice(@sdlfile,$i-2,0,”/* CIF Label (350,100) (100,100) */\n”);
          $i++;
          splice(@sdlfile,$i-2,0,”connection $stateName;\n”);
          $i++;
}

        #$i++;
        $statePosX = 400; $statePosY = 200;

        # Find the place in the CIF file header where the pages are declared
        $pagedeclare = 0;
        while(($sdlfile[$pagedeclare] !~ /CIF Page $lastName/i))
        {$pagedeclare++;
        }

        # Insert declaration of the new page in the CIF header
        splice(@sdlfile,$pagedeclare+2,0,
        ”/* CIF Page $pageName\part$newPages ($pageWidth,$pageHeight) */\n”);
        splice(@sdlfile,$pagedeclare+3,0,
        ”/* CIF Frame (0,0),($pageWidth,$pageHeight) */\n”);
        $pagedeclare = $pagedeclare+2;
        $i=$i+2;
        #Skip two lines, since the declaration has moved everything downwards
        $lastName = ”$pageName\part$newPages”;
      }

     #Check horizontal alignment
     $rightX = $flowsUsed * $flowSpacing + 300; print ”RightX: $rightX\n”;
     if($rightX != $symbolX)
     { # Move the flowline to the right x-coordinate

      #Check the line from the state symbol to the input symbol
      if($sdlfile[$i-2] =~ /CIF Line \((\d+),(\d+)/i)
        { unless($1 == $statePosX && $2 == statePosY)
  {$tmp = sprintf(“/* CIF Line (%d,%d),(%d,%d),(%d,%d),(%d,%d) */\n”,
   $statePosX,$statePosY,$statePosX,$statePosY+25,$rightX+100,
   $statePosY+25,$rightX+100,$symbolY);
          splice(@sdlfile,$i-2,1,$tmp); #print ”$tmp”;
          }
          #print ”$1 $2”;
        }
       else {print ”Couldn’t find line to symbol in $sdlfile[$i-2]”;}

       # Evaluate offset
       $x = $symbolX- $rightX;
       print ”Moving a flowline in $stateName $x pixels to the left\n”;
       print ”at line $i : $sdlfile[$i]”;
       $i--; # Step one line up to start with the CIF signal
       $startDecide = $i; $flowsDecide = $flowsUsed;
       while($sdlfile[$i+1] !~ /^\/\* CIF (Input|Save)/i &&
             $sdlfile[$i] !~ /^(endstate|endconnection)/i)
       {
         #Correct the x-coordiantes in the flowline
         # In a CIF label or Join statement, only the first x must be changed,
         # since the second x defines the size of the symbol.
         if($sdlfile[$i] =~ /\/\* CIF (Label|Join|Comment|Text) /i)
         { $sdlfile[$i]=~ s/\((\d+)/$1-$x/e;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2/;
           # Put the parantheses back in
         }

         elsif($sdlfile[$i] =~ /\/\* CIF/)
         { $sdlfile[$i]=~ s/\((\d+)/$1-$x/ge;
           # Substitute ’(x’ with x minus offset
           $sdlfile[$i]=~ s/(\d+),(\d+)/\($1,$2/g;
           # Put the parantheses back in
         }
  if($sdlfile[$i] =~ /^(nextstate|join)/) {$flowsUsed++;}
        $i++;
        } # End adjusting x-coordinates

     } # End fix alignment
    } # End check input signal

    if($spaceNeeded > 1)
    {  # Check spacing inside decisions
       $j = $startDecide;
      while($sdlfile[$j+1] !~ /^\/\* CIF (Input|Save)/i &&
           ($sdlfile[$j+1] !~ /(endstate|endconnection)/i) && $j < @sdlfile)
       {
        if($sdlfile[$j] =~ /^(join|nextstate)/i &&
          ($sdlfile[$j+1] =~ /CIF Answer/i) &&
          ($sdlfile[$j+2] =~ /\((\d+),(\d+)\) \*\//))
        {
         $flowsDecide++;
         $x = $1 -100;
          print ”Inside! x = $x, flows = $flowsDecide, $sdlfile[$j+2]”;
          #Check horizontal alignment
          $rightX = $flowsDecide * $flowSpacing + 300;
          #print ”RightX: $rightX\n”;
      if($x != $rightX)
      { # Evaluate offset
        $x = $x - $rightX;
        print ”Moving a flowline Inside a decision in
               $stateName $x pixels to the left\n”;
        print ”at line $j : $sdlfile[$j]”;

       while($sdlfile[$j+1] !~ /^(join|nextstate)/i)
       {
         #Correct the x-coordiantes in the flowline
         # In a CIF label or Join statement, only the first x must be changed,
         # since the second x defines the size of the symbol.
         if($sdlfile[$j] =~ /\/\* CIF (Label|Join|Comment|Text) /i)
         { $sdlfile[$j]=~ s/\((\d+)/$1-$x/e;
           # Substitute '(x' with x minus offset
           $sdlfile[$j]=~ s/(\d+),(\d+)/\($1,$2/;
           # Put the parantheses back in
         }

         elsif($sdlfile[$j] =~ /\/\* CIF/)
         { $sdlfile[$j]=~ s/\((\d+)/$1-$x/ge;
           # Substitute '(x' with x minus offset
           $sdlfile[$j]=~ s/(\d+),(\d+)/\($1,$2/g;
           # Put the parantheses back in
         }

        $j++;
        } # End adjusting x-coordinates
       } # End x != rightX
      } # End found a nextstate/join
        $j++;
     } # End searching the flowline
    } # End checking inside decisions

    $i++;
   } # End search through page

  } # End check page
  else {$i++;}
} # End search through sdlfile
} # End procedure

sub removeEmptyPages
{ # Purpose: Remove pages that have become empty after deletion of connections
  $i = 0;

  while($i < @sdlfile)
  { if($sdlfile[$i+1] =~ /^\/\* CIF currentPage/i &&
      $sdlfile[$i] =~ /^\/\* CIF currentPage (\w+)/i)
      { # Delete the pagebreak
        $pageName = $1;
        print "At line: $i Deleting pagebreak $pageName\n";
        splice(@sdlfile,$i,1);
       }
      $i++;
  } # End removing unnecessary pagebreaks

  # Check if the declared pages are in fact used
  $i = 2;

  while($sdlfile[$i] =~ /^\/\* CIF (Frame|Page)/i)
  { if($sdlfile[$i] =~ /^\/\* CIF Page (\w+)/i)
    { $pageName = $1; #print ”Checking page $pageName\n”;
      $inUse = 0;
      $j = $i;

      while(($j < @sdlfile) && ($inUse == 0))
      { if($sdlfile[$j] =~ /^\/\* CIF currentPage $pageName/i)
          {$inUse = 1;}
        $j++;
      }

      # If not in use, Delete the declaration
      if($inUse == 0)
      {  splice(@sdlfile,$i,1);
         splice(@sdlfile,$i,1);
         print ”Deleted declaration of page $pageName\n”;
      }

     } # End chech declaration
    $i++;
  } # End removing unused declarations
}
```