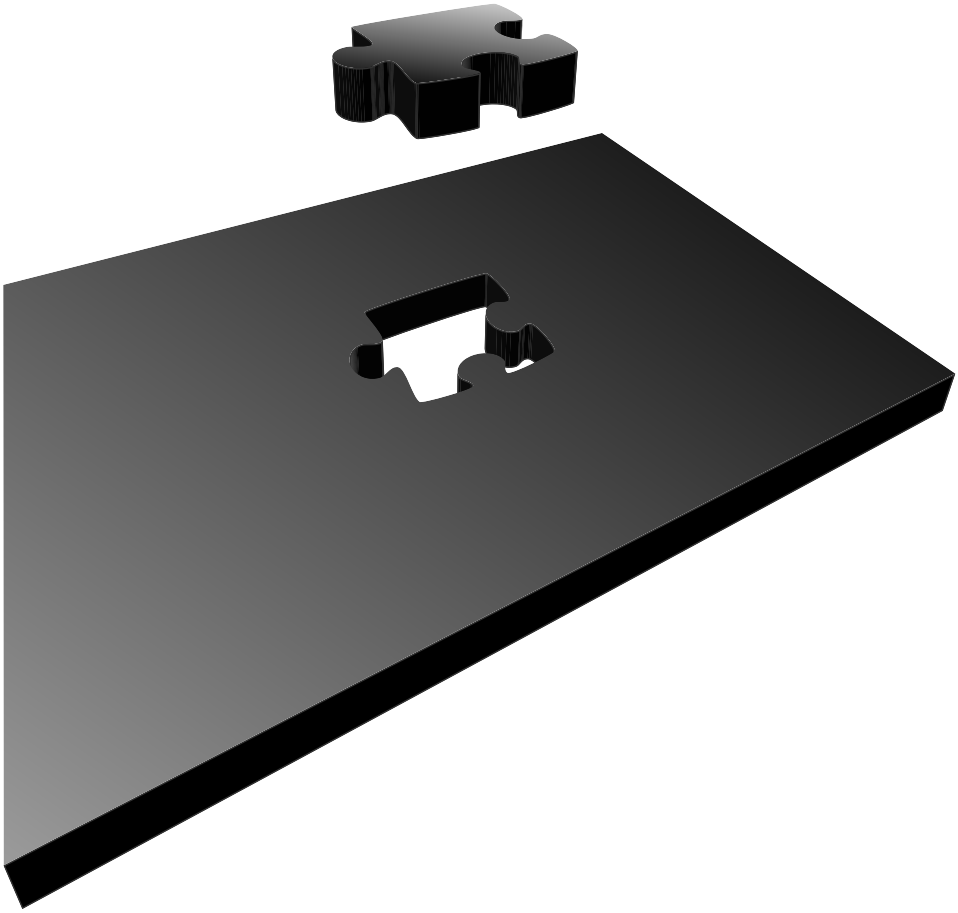
Logical Expressions: Analyzing, Generalizing, Rewriting

embedding a scientific implementation in a commercial environment



Alexander van den Bergh
1999

Preface

The Plan

December 1998 I finished my last tests and was only a master thesis away from graduating. After four and a half years at the University of Amsterdam I thought it was time to take a look in the corporate world. Since I had no idea where to apply for a scientific internship I asked Paul Klint whether he had any options. After a few searches and some rejections I ended up at TriLoc Software Engineering.

TriLoc had a close relationship with the university and needed someone to experiment with the technology they were buying from them. It turned out the technology was mostly ASF+SDF with which I was already acquainted.

We formulated a project in which it was my task to see whether the ASF+SDF rewriting capabilities could be useful for TriLoc and if it would be possible to embed them in their software renovation processes.

The university agreed with the plan and although nobody (including myself) thought the project would be completed in the given time, we started the implementation.

The Outcome

The environment at TriLoc turned out to be quite good. They provided me with some nice equipment, a Sun workstation (Sparc 5) for the Meta-Environment, an IBM PC (266 Pentium II) for C development and porting issues and of course I had my own Apple Powerbook and G3 desktop to make the platform family complete.

I must have gained at least ten pounds during the internship thanks to the very tasty (and cheap) lunches and the numerous birthday pies I ate. I had the freedom to visit the university and the library whenever I considered it useful.

After a little more than four months the practical side of the project was completed. We even went further than the actual plans. As I write this, the ASF+SDF technology is ported to Windows NT and practically ready to be used within the TriLoc system. A very comprehensive COBOL logical expression rewriter is implemented and tested on COBOL code from ‘real life’ systems.

Acknowledgments

One of the main reasons that the project succeeded in such a small period of time is the tremendous feedback I received from both the University of Amsterdam, the National Research Institute for Mathematics and Computer Science (CWI) and TriLoc.

At all sides people were willing to cooperate with the project. Without their feedback it would definitely have turned into a time-consuming frustrating project. I would like to thank a few of them by name.

First of all, I would like to thank Mark van den Brand and Gert Veltink for their support on all domains. Gert especially helped me get started and trace the hard to find bugs in the ATerm library sources while Mark clarified numerous issues regarding the whole Meta-Environment and everything that comes with it. Furthermore a lot of credits go to Jeroen Scheerder for providing support on the parser, Tobias Kuipers for reacting on my phone calls and emails, Pieter Olivier for his help with the ATerm-library and Pieter Bloemendaal for his cooperation from TriLoc’s side.

The Meta-Environment support team: Jeroen, Tobias and Mark; their ears still red from the many phonecalls I made to them.

Paul Klint gave me general directions, without him, Ger Bakker, Gert-Jan Tretmans and other people from the TriLoc management team this project never would have existed in the first place. Lia Bos and Lucienne Evers provided me with information on Logic Mining and Jacob Brunekreef supported me on the ASF+SDF development and this thesis.

Although all these people helped me a great deal and I have got enough talent of my own, none of this would have even come close to success if it wasn't for the unconditional support of my parents. So last but definitely not least I would like to thank them, George and Mientje, for supporting me.

When reading this thesis you might find the layout to be a little different from what is expected from a master thesis.

I have tried to make this thesis both pleasing for the mind as for the eye. I hope you enjoy reading it.

Yours truly,

Alex

Photography by Remko Schnorr

Disce ut semper victurus, vive ut cras moriturus.

Logical Expressions: Analyzing, Generalizing, Rewriting

embedding a scientific implementation in a commercial environment

Alexander van den Bergh

Afstudeer docent
prof. dr Paul Klint

Begeleiders
dr Mark van den Brand
dr Gert Veltink

April - September 1999

1. Introduction

This chapter serves as a general introduction to the remainder of this thesis. We will try to put the following parts into context by describing the need for this research.

Although one could argue that the scenario presented in this cartoon is a little pessimistic. There still is a lot of uncertainty about what is going to happen when the Year 2000 Bug is going to 'hit our planet'.

1.1 The Software Crisis

During the past decade software renovation has become an important field in computer science. Due to large problems like Y2K and the new European currency, the Euro, billions of dollars have been and will be invested in projects to analyze and renovate existing systems.

Legacy Systems

The development of larger software systems started decades ago, their source code grew over the years and because of that they have become an unmanageable chaos of code from a mixture of different dialects of programming languages. These systems are referred to as *legacy systems*. Legacy systems, and their complexity, have presented us with the undesirable situation of extremely hard to manage software systems which has resulted in a new *Software Crisis**.

Currently software has become the backbone of most companies. Since almost everything is computerized, it has become an important source to describe the processes within a corporation. Unfortunately there is very little knowledge about these software systems, which have become legacy systems over time, therefore the adjustment of them can be a time consuming and painful process for software analysts. Special task forces of analysts are full-time operational to maintain these systems. To help these analysts with their analysis and renovation, specific software renovation systems have been developed. Some platforms for software renovation factories are TXL [CHP91], COSMOS [Em98] and REFINE [RA92].

The lack of knowledge about the legacy systems does not only cause large technical problems (such as the Y2K problem) but it also enlarges the time-to-market. In a fast moving market, companies need to be able to react quickly to changes, with unmanageable legacy systems it takes too much time, it is too expensive and it involves taking too many risks. Competitiveness means flexibility and speed when it comes to new services and products.

* The situation referred to as *the* software crisis started when people experienced that correct software could not be engineered like bridges and buildings, first designing it on paper and then creating it without any errors.

What to do?

So why not invest a little money and start building a new, clean, modular, and easy to understand system? This might seem an option from a nonprofessional point of view, but when looking closer at the state of affairs within the software crisis we see that this is not realistic.

First of all these systems are big, not to say huge. It will cost more then just 'a little' money, it will require enormous amounts of money, time and manpower. It means that the investment in the current system, which could easily have reached billions of dollars, is practically thrown away. Some companies (like banks) might be able to finance at least the first part of the project, however the time and manpower will often not be available.

A second thing that is overlooked by the 'start again from scratch' option is the fact that virtually all company knowledge is hidden within the source code of the old system. This means that the new system must support all features of the old one, including the ones that were hidden. Unfortunately there often is no other source of information, such as reliable documentation, describing the behavior of the old system, which makes a re-implementation impossible.

The only option seems to be to keep the current sources and to retrieve as much information from them as possible.

Logic Mining

The *Logic Mining* technique is initiated, developed and proclaimed by TriLoc Software Engineering. This company proposes a new revolutionary approach on software renovation. Their strategy is build around the idea of extracting (mining) information (logic) from legacy systems. The approach focuses on the business logic rather than the technical ground of the system. It sees the legacy system as an important asset with reusable value for the future instead of a millstone around one's neck.

This in-depth analysis done by Logic Mining can be helpful in various situations such as isolating components that perform certain tasks (for maintenance), code slicing techniques, identifying business processes, and modelling or documenting the system.

Term rewriting

Analysis and transformation of existing code form the core functionality of software renovation systems. When using formal methods, like term rewriting, for these tasks, we work with a firm mathematical basis for the system. To *describe* a language in an imperative programming language like C we need to write a parser for it. Directly writing a parser for a certain language can quickly become a large project and therefore hard to debug. In a formalism like SDF we *define* a grammar without having to worry about parsing techniques and algorithms.

The usage of compiler construction techniques such as scanning, parsing, typechecking and analysis can be useful in the field of re-engineering as described in [BKV96-b]. Systems that support specification formalisms nevertheless are not widely available. And even if we have a formalism at our disposal we face serious technology transfer problems when we try to sell term rewriting to the industry [BKV98].

1.2 The Project

TriLoc, however, has taken the challenge to investigate whether state of the art scientific technology can be adapted to be useful within the Logic Mining process. In close cooperation with the University of Amsterdam and the National Research Institute for Mathematics and Computer Science (CWI) a project was initiated to accomplish this.

The project contains four subprojects:

1. Describing a rewriter in the algebraic specification formalism ASF+SDF that can be used to normalize COBOL logical expressions.
2. Defining other rewriters and grammars including a language independent logical format on which various normalization techniques can be applied.
3. Porting the rewriting component (that supports execution of ASF+SDF rewriting systems) of the ASF+SDF Meta-Environment to Windows NT.
4. Embedding its functionality in the Logic Mining system.

We will describe the process in two parts which can be read independently:

1. The analysis, normalization, generalization and rewriting of (COBOL) logical expressions.
2. Porting parts of the ASF+SDF Meta-Environment to a non-Unix environment and embedding these parts in an external commercial software renovation system.

A key feature of this project is the communication between science and practice:

- the cooperation between TriLoc on one side and the University and the CWI on the other, the port of a Unix-system to Windows NT,
- the embedding of a scientific system within a commercial system and
- the extending of a system made with conventional languages with a system designed with an algebraic formalism.

UNIVERSITEIT VAN AMSTERDAM

1.3 Related Work

There have been projects that focussed on rewriting COBOL. We will name a few here. Control flow normalization, in order to improve maintainability of COBOL/CICS systems, which also makes use of normalized logical expressions is described in [BSV98]. Normalization of COBOL conditions is also a subject in [SV98]. In [BKV98] a global description regarding the rewriting of COBOL systems is given. Issues regarding COBOL grammar in SDF, disambiguation, preprocessing and restricting can be found in [BSV97].

In this thesis we will try to be more thorough. We will introduce a language independent format for logical expressions and normalizations on this language independent format.

In [BKV96] possible software re-engineering applications of the Meta-Environment technologies are described. A software renovation factory is described in [BSV98]. It contains a number of assembly lines that perform a number of tasks in a fixed order. Our project is not quite the same since it integrates a component in a system on a more object-oriented way, it can be called at any given time from another process.

Part I

A General Model for Logical Expressions

In the field of software re-engineering higher levels of abstraction are needed in order to analyze existing systems. Source code must be analyzed and visualized in a clear and unambiguous way.

This part discusses a language independent format for logical expressions, the General Logical Format, GLF, that is used in a re-engineering environment. A rewriting system for translating COBOL logical expressions into this independent format will be presented as well as various transformations on GLF.

2. Introduction

In this chapter we will explain why we feel that rewriting of logical expressions is necessary. We will describe some practical issues as well as some excentric properties of logical expressions in COBOL [COB].

Softwae development should, just like other large construction work, be done by engineers

2.1 The Code

An important step in software analysis and renovation is the analysis of source code. Although documentation of the code can be of help it is often out of date and not in-sync with the actual programs. The only way to be certain that you are indeed analyzing the most up-to-date information is by analyzing the actual source code.

Conditional and Logical Expressions

Code can be divided into various classes. We can identify data, assignment statements, input/output code, subroutines, conditional expressions and many more. All these items have their own effect on the behavior of the program. In this thesis we will focus on the logical expressions that are the most important part of conditional expressions.

Conditional expressions are important when analyzing the control flow, which is of interest when trying to understand the behavior of a program. As soon as there is any form of interaction with users or data you will find conditional expressions that act according to the values of logical expressions. An example of a basic conditional expression:

```
If <Logical Expression> then
     <Statement 1>
else
     <Statement 2>
endif
```

In our case a logical expression consists of one or more relational expressions separated by logical operators. For instance:

```
SALDO > 0 AND PINCODE = 3675
```

is a valid logical expression since it consists of the relational expression `SALDO > 0`, the relational expression `PINCODE = 3675` and the logical operator `AND`.

Clear and readable logical expressions are of great importance for software analysts. Logical expressions can be written in various ways. They can be rewritten to other notations to be more readable, easier to analyze or easier to classify without losing their information and their behavior.

The Problem: too Many Different Notations

Although they mostly stick to rather simple propositional logic, many discrepancies between logical expressions in various programming languages exist. In Pascal [Wirth71, SAV91] we can for instance write `(A = B)` which yields TRUE when the value of A equals the value of B whereas in C [KR88] `(A = B)` means that the value of B is assigned to the variable A. This expression yields the value stored in B (which can be either zero (regarded FALSE) or non-zero (TRUE)). The correct C equivalent of the Pascal statement is `( A == B )`.

Some languages allow notations that are intuitively hard to understand. For instance in COBOL [COB, Ebb90] one can write `A = B OR C` which can also be written in a clearer way: `( A = B ) OR ( A = C )`. A more complex example is `A equals B OR C OR NOT D AND is greater than E OR F` which is equivalent to `(A = B) OR (A = C) OR ( not ( A = D ) AND ( A > E ) ) OR (A > F )`. Analyzing and rewriting all these different notations is a complex task. For human analysts it is hard to interpret those large COBOL expressions without making mistakes. For computer programmers a system that can rewrite logical expressions for all these languages can become too large to create, verify and manage.

The Solution: a Universal Notation

Although the notations differ, most logical expressions are based on the same propositional logic. The solution we present is to translate the various notations of logical expressions of a language (or more languages) into a generalized form. This notation will be unambiguous, compact, language independent and easy to interpret by a computer.

We call this notation the *Generalized Logical Format*, or just *GLF*. Expressions written in GLF can then be transformed into normal forms, a more (human-) readable form or perhaps even back to a (different) programming language.

The Approach: Step by Step

The process we will describe in this thesis is the translation of COBOL logical expressions to GLF from which they can be translated back to a readable representation. The benefit of translating different languages to GLF first, gives us the ability to use only one rewriter for the next rewrite operations. This not only saves us a lot of development work but it also provides us with a very flexible system as shown by Figure 2.1.1.

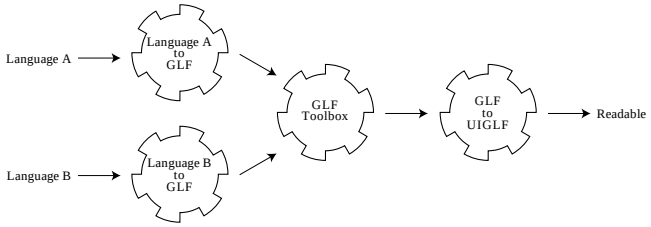


Figure 2.1.1

Figure 2.2.1 shows that only one part of the system is (input-) language dependent. The other transformations simply work on GLF. Before we translate GLF to readable expressions (which we will later call *UIGLF*) and we can perform additional GLF transformations on them. Those transformations can be adjusted, extended or deleted without having influence on the original input programming languages.

When a system like this has been developed for language 'A' we simply have to make one rewriter that translates language 'B' to GLF and all the existing GLF-operations can be reused for this language as well. If a translator to GLF for a given language exist the rewritings on GLF can be regarded *language parameterized* (or *generic*).

GLF can also be valuable to code analysts since it clarifies the meaning of the logical expression.

Terminology

We identify three types of *rewriters* in our process. When working with COBOL we will see that *disabbreviation* is needed. After that, a *translation* will translate our COBOL to GLF on which we can perform various *transformations*. All of these rewritings can be done using an ASF+SDF *term rewriting system (TRS)*.

2.2 ASF+SDF

To specify the grammar of our source and target languages and to specify the behavior of our rewriting systems we will use the ASF + SDF formalism. The Algebraic Specification Formalism, ASF [BHK89], provides us with the ability to define the abstract syntax (i.e. type and arity) of functions (rewrite rules) and write conditional equations defining the semantics of these functions. The Syntax Definition Formalism [HKR89], SDF, allows the integrated definition of lexical, context-free and abstract syntax. It contains enough information to generate parse tables for the specified syntax. Together they form ASF+SDF specifications.

Why ASF+SDF?

A number of formalisms exist in which it is possible to define programming languages. Systems like the well known Lex+Yacc duo [LS86, John75] and Refine [Alto92] also offer parsing and rewriting techniques. ASF+SDF with its Meta-Environment [BKM97] has the advantage of providing modularity. Modularity is of great importance considering the size and extendibility of our project. Furthermore the LR-parsing done by Refine and Lex+Yacc is less thorough than SGLR-parsing [Vis97-2] done by ASF+SDF since the former may offer shift-reduce and reduce-reduce conflicts [BSV97].

In part II of this thesis we will give a more detailed description of the ASF+SDF Meta-Environment, also describing its AsFix to C compiler which we will be needing in the future. Finally the Meta-Environment is academic rather than commercial which makes it a free product for universities and research institutes.

3. COBOL Logical Expressions

The main focus of our project will be to work on COBOL logical expressions. This chapter explains why this language is especially eligible for our purposes.

Grace Murray Hopper a mathematician and pioneer in data processing. An expert on compilers and a key figure in the design of COBOL. Legendary among both computer scientists and industrial executives.

3.1 COBOL

In the late 50's computer manufacturers and users saw the need for a more universal way to program computers. A group of experts was brought together to develop a programming language that be easy to write, read and maintain, ideal for writing business applications and be compatible with all computers. The language *COBOL* [Ebb90], which stands for COmmon Business Oriented Language, was born. It indeed became the most commonly used programming language for data processing which is often done in business applications.

Many different dialects have been developed for COBOL through the years. Now, in the late 90's, many companies are dealing with millions of lines of old legacy COBOL code within their software. Analysis and maintenance of this code is hard since it often contains code written in outdated dialects, with hardly any up-to-date documentation and without a clear and consequent style.

COBOL Logical Expressions

COBOL [COB] does not put many restrictions on programmers to write clear code. Logical expressions can be written in numerous ways which can be quite confusing. With the English language in the back of their minds the designers of COBOL allowed the programmer to write a simple relational expression (in which two operands are compared) in many different ways:

```
A = B
A IS B
A IS EQUAL B
A IS EQUAL TO B
```

Of course, notations like these can easily be translated to one preferred notation; this can be done in a preprocessing phase thereby reducing the many alternatives in production rules of the context-free grammar. In the remainder of this thesis we shall only be using the 'mathematical' notation (A = B) rather than the 'English' notation.

Another somewhat special notation can exist because of the fact that COBOL allows the embedding of the logical operator 'NOT' (the negation) within relational expressions.

This means that all the following expressions have the same meaning:

```
A <> B
A NOT = B
NOT A = B   (this must be read like NOT ( A = B ) )
```

Note that the first expression is a relational expression, the last is a logical expression (negated relational expression) and the middle one is a logical expression where the logical operator is placed within the relational expression.

The biggest problem with COBOL conditions, however, are the abbreviated conditions. This allows programmers to leave consecutive operands and/or operators out of certain logical expressions. This will be illustrated by a couple of examples.

The abbreviated expression is quite readable, the variable is being compared to a set of possible values. The operator (=) and the operand (A) are implicitly applied to the rest of the operands in the expression:

```
A = B OR     C OR     D
A = B OR A = C OR A = D
```

In the following example the operand propagates through the expression to the relational expressions C, D and E. It can, however, not propagate straightforward because the third relational expression (> D) has its own operator which propagates to the last relational expression. When a new operator is encountered we proceed with that one:

```
A = B OR     C OR   > D OR     E
A = B OR A = C OR A > D OR A > E
```

When a 'NOT' is encountered right before an operand (without an operator) that relational expression is negated. The 'NOT' does not propagate further in the expression. The brackets in the expression are only added to improve readability:

```
A = B OR NOT      C  OR     D
A = B OR NOT (A = C) OR A = D
```

When a 'NOT' is encountered before an operator, the operator as well as the 'NOT' propagate through the rest of the expression:

```
A = B OR NOT    > C  OR          D
A = B OR NOT (A > C) OR NOT (A > D)
```

Abbreviated expressions do not care about the priorities of logical operators. One can argue that since conjunction binds stronger than disjunction the '<'-operator should not propagate outside the conjunction (making the following expression equal to `A = B OR A = C AND A < D OR A = E`) but that is not the case. The (dis-)abbreviation of logical expressions is more a textual than a logical process:

```
A = B OR     C AND   < D OR     E
A = B OR A = C AND A < D OR A < E
```

Combining some of the properties of COBOL we have seen so far, the following two expressions have the same meaning:

```
A NOT = B  OR  NOT C  AND  B = D  AND    > E   OR      F
NOT(A = B) OR (A = C) AND (B = D) AND (B > E ) OR (B > F)
```

A COBOL programmer can choose any style in which he or she wants to program. Styles can also be mixed leading to unreadable expressions.

COBOL also allows 88-fields within logical expressions. These fields are actually tests whether a variable (or field within a record) has a certain predefined value. The system we present does not pay attention to them, in our case they have been modified to relational expressions by a preprocessing phase:

Consider the record

```
01 TEST-RECORD.
  03 STATUS PIC 9.
    05 END-OF-LINE VALUE '1'.
    05 END-OF-PAGE VALUE '2'.
    05 END-OF-FILE VALUE '3'.
```

The conditional expression

```
IF END-OF-LINE THEN
```

is translated by the preprocessor to (the '::' is a record-field notation: `record::field`)

```
IF TEST-RECORD::STATUS == TEST-RECORD::STATUS::END-OF-LINE
```

Priorities in COBOL

To for ce the priorities within arithmetic operations in COBOL we have designed our COBOL grammar in SDF in such a way that the parse tree already contains these priorities. Arithmetic operations (on operands) are typed. Operands of an operation can be of the same type or they have to be of a type of an arithmetic operation with a higher priority. We will illustrate this by taking a look at the arithmetical expr essions in COBOL. For instance a multiplication can take other multiplications as operands, it can take an involution as an operand but it can not take an addition as an operand.

```
CB-PowerOfExpr CB-MulSubExpr*      -> CB-MulExpr
CB-UnaryExpr   CB-PowerOfSubExpr*  -> CB-PowerOfExpr

"*"   CB-PowerOfExpr               -> CB-MulSubExpr
"/"   CB-PowerOfExpr               -> CB-MulSubExpr

"^"   CB-UnaryExpr                 -> CB-PowerOfSubExpr
```

A number or an identifier has the highest priority and a substraction has the lowest priority.

Rules similar to this should apply for relational and logical operators as well. Unfortunately COBOL's abbr eviated expr essions cause pr oblems, because the parse tr ees for these abbreviated expressions can not be evaluated or translated because they do not contain all the needed information. Consider the COBOL expression `A < B AND C` when we would be using our priority forcing types we would create the parse tree shown in Figure 3.1.1.

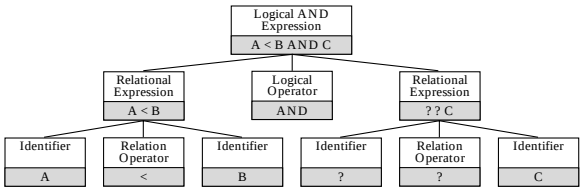


Figure 3.1.1

The right Relational Expression subtree does not contain all the information for translation into 'A < C'. The information we need, the operand (A) and the operator (<), is higher in the tree up to the level of the 'Logical AND Expression'.

3.2 Clean COBOL

Our first term rewriting system translated COBOL directly into GLF. Although it worked correctly we decided to take another approach since the specification had become too complex to analyze, verify and adjust.

Most other programming languages do not allow abbreviated conditions which makes the process of translating to GLF less complex. Fortunately all abbreviated conditions can be *disabbreviated* to their non-abbreviated counterpart. To make the specification of our rewriter more modular and easier to understand we decided to split the COBOL to GLF translation process in two phases.

We will first *disabbreviate* COBOL to Clean COBOL, then we will *translate* Clean COBOL to the Generalized Logical Format. Clean COBOL does not allow abbreviated expressions, negations within relational expressions, redundant (double) 'NOT'-operators or "English synonyms" for relational operators. Clean COBOL is 100% pure COBOL but written down in a similar way modern languages are written down.

Within Clean COBOL we work with simple relational expressions consisting of two arithmetic expressions (consisting of identifiers, constant numbers or mathematical expressions) with a binary relational operator (such as =, > or =<).

On a higher level we work with Unary Logical Expressions where we can take a relational expression and put an optional 'NOT' before it (or we can take a Logical Or-Expression and put brackets around it, with an optional 'NOT').

These Unary Logical Expressions are valid Logical And-Expressions which may be followed by the logical operator 'AND' (conjunction) and another Unary Logical Expression.

Clean COBOL can is a standardized way to write down logical expressions. It can be regarded a *canonical form* of COBOL.

Finally, we have Logical Or-Expressions which consist of a Logical And-Expression optionally followed by the logical operator 'OR' (disjunction) and another Logical And-Expression.

A partial parse tree of `NOT A > B AND A < C OR A = 0` looks like Figure 3.2.1.

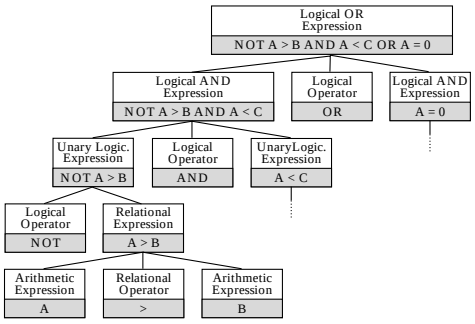


Figure 3.2.1

New priorities in Clean COBOL

The parse tree illustrates the priority rules that apply to logic: first the negation, then the conjunction and finally the disjunction. Since we do not work with 'NOT'-operators within relational expressions we can make our types more simple. This parse tree is a clear top-down representation of the actual logical expression. Any subtree is a valid COBOL expression.

The priorities of the (logical) operators in Clean COBOL are forced by the hierarchical structure of the grammar. The sorts are ordered in such a way that they force certain language constructs (with a higher priority) to be low in the parse tree, while others exist on a higher level (lower priority).

Clean COBOL in SDF

Clean COBOL has no explicit definition in SDF. It is merely a subset of the normal COBOL grammar. Where COBOL allows abbreviated relational expressions and 'NOT's in relational expressions Clean COBOL only focuses on their clean counterparts:

```
context-free syntax
  CB-RelExprOperator              -> CB-OpNot
  "NOT" CB-RelExprOperator        -> CB-OpNot

%% Clean COBOL and COBOL
  CB-AddExpr CB-RelExprOperator CB-AddExpr       -> CB-RelExpr

%% (normal) COBOL only
  CB-AddExpr CB-OpNot CB-AddExpr   -> CB-RelExpr
             CB-OpNot CB-AddExpr   -> CB-RelExpr
                      CB-AddExpr   -> CB-RelExpr
```

The Logical expressions have the same grammar. Notice how priorities are forced with the use of sorts and subsorts:

```
%% Highest priority, relational expressions and brackets.
  CB-RelExpr                      -> CB-UnaryLE
  "(" CB-LogOrExpr ")"            -> CB-UnaryLE

  "NOT" CB-UnaryLE                -> CB-UnaryLE

  "AND" CB-UnaryLE                -> CB-AndShort
  CB-AndShort*                    -> CB-AndShortList
  CB-UnaryLE CB-AndShortList      -> CB-LogAndExpr

%% Lowest priority, the disjunction 'OR'
  "OR" CB-LogAndExpr              -> CB-OrShort
  CB-OrShort*                     -> CB-OrShortList
  CB-LogAndExpr CB-OrShortList    -> CB-LogOrExpr
```

The Cleanup disabbreviater takes and returns COBOL:

```
%% CleanUp fixes abbreviated relexpressions, removes redundant comma's
%% moves NOT's within relexpr to their UnaryLE's
%% and removes double NOT's
  "CB-CleanUp" "(" CB-LogOrExpr")" -> CB-LogOrExpr
```

4. Generalized Logical Format

Logical Expressions in programming languages are much alike. Therefore it is possible to write them in a language independent format. This chapter will introduce the Generalized Logical Format, GLF, and its more readable form UIGLF.

```
CB2UIGLF-Condition(
    A = 02 OR 03 OR
    ( A > 11 AND A < 17 )
    OR
    ( A > 19 AND A < 25 )
, CNF-NOT )

( ( A == 02 ) or ( A == 03 ) or ( A > 11 ) or ( A > 19 ) )
and
( ( A == 02 ) or ( A == 03 ) or ( A > 11 ) or ( A < 25 ) )
and
( ( A == 02 ) or ( A == 03 ) or ( A < 17 ) or ( A > 19 ) )
and
( ( A == 02 ) or ( A == 03 ) or ( A < 17 ) or ( A < 25 ) )
```

A fragment of in- and output of the first 'real system' that was written by the TRS. The input is COBOL, the output is UIGLF in CNF without negations.

4.1 GLF

With GLF we can write logical expressions in a language independent way that is easy to describe, parse and rewrite.

The Logic of GLF

When describing the logical operators used in GLF we regard it as a propositional language *L* containing the following symbols

- a set of propositional variables *Var-L*, these are our relational expressions [KN97];
- the unary logical connective: 'not(F)', the NOT F;
- the binary logical connective: 'or(F,G)', the F OR G;
- and defined by the above the binary logical connective: and(F,G)
  which abbreviates: not(or(not(F),not(G))).

All the formulas over *Var-L*, denoted as *Form-L*, are defined as follows:

- An element of *Var-L* is a formula over *Var-L*
- If *F* and *G* are formulas over *Var-L* then not(F), called negation, and or(F,G), called disjunction, are formulas over *Var-L*.

We can define the well known semantics of these operations with the Boolean set bools {true, false} and a valuation-function v: Form -> bools which satisfies:

v(not(F)) = false if v(F) = true
v(not(F)) = true if v(F) = false
v(or(F,G)) = false if v(F) = false and v(G) = false
v(or(F,G)) = true if v(F) = true or v(G) = true

Although we will not be evaluating the logical expressions in this thesis we will need to know their semantics in order to make the right decisions in future rewritings, for instance it is not hard to see that not(not(F)) has the same semantics as F.

Tags in GLF

GLF labels certain tokens with a type-tag. This is done to prevent loss of information. Here are some examples:

i(4)	decimal number (integer) 4
h(4)	hexadecimal number 4
r(4)	real number 4

Tokens can get more than one tag because of the use of chain functions within SDF (type1 is of type2):

vn(BLA)	variable name BLA
id(vn(BLA))	identifier , variable name BLA

Since operators in GLF are written in prefix they do not get an extra tag:

add(i(1),i(1))	also known as 1+1
add(mul(i(1),i(1)),i(1))	can be written as 1*1+1

This assures us that every bit of code in GLF is preceded by a tag describing its meaning.

Functions and Priorities in GLF

To keep things compact and simple we have chosen to use functions with a maximal arity of 2. Because of this and of the prefix notation we do not have to worry about associativity .

The arithmetical expression	'1 * 2 * 3'
can be interpreted as	'(1 * 2) * 3' or '1* (2 * 3)'
whereas	mul(mul(i(1),i(2)),i(3))
can only be interpreted as	'(1 * 2) * 3'.

Priorities are also implicitly added and brackets are obsolete.

The expression	'(1 + 2) * 3'
can only be written as	'mul(add(i(1),i(2)),i(3))'
and	'1 + 2 * 3'
leads to	'add(i(1),mul(i(2),i(3)))'.

In the COBOL grammar we defined our priorities by means of types. With GLF we cannot do that. Since the prefix notation allows each operator to have operands of any priority without the use of brackets. We work with 'arithmetic expressions' only , rather than with 'addition expressions', 'multiplication expressions' and 'power expressions'.

For instance the GLF multiplication can have two arithmetic arguments without any constraints on the (operation that is in effect on its) arguments. Arithmetic operations form expressions of the same type and they can all be operands for (other) arithmetic operators:

```
"add" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"sub" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"mul" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"div" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"pow" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
```

4.2 Untyped Infix GLF

Although GLF is easy to interpret from a computer/parser perspective, larger expressions are hard to read for humans. Human analysts who work with GLF need a more readable form. To meet those demands we introduce the *Untyped Infix GLF*, or just *UIGLF* (since its mainly purpose is to improve readability for users it can be regarded 'User Interface'-GLF as well).

UIGLF is GLF translated to an infix form, without the type-tags, with extra brackets to improve readability and with a small set of easy to interpret keywords.

The 'Untyped' in UIGLF

Untyped of course means the removal of the tags. Removal of tags also means removal of some information but in this case we assume the analyst has knowledge of the context of the code, such as the programming language so, he or she can differentiate between variables and keywords. On the other hand the removal of information makes the expression easier to understand.

The 'Infix' in UIGLF

We have chosen infix instead of prefix notation when it comes to readability. In natural as well as mathematical notation this is the way to go:

4 - 2 is written and pronounced in infix (four minus two).

The alert reader will notice that this also means that in some cases brackets have to be inserted:

mul(4, add(2,3)) can not be written as	4 * 2 + 3
it must be written with brackets around the addition:	4 * (2 + 3).

The 'User Interface' in UIGLF

Other readability issues are addressed within UIGLF as well. The priority rules for instance are written down with brackets to make the analyst aware of them:

	4 + 3 * 2
is written down as	4 + (3 * 2)
and	A OR B AND C
is written down as	A OR (B AND C)

This feature can cause undesired effects. Too many brackets can decrease readability:

(A AND (B AND (C AND D))) OR E

We find that the brackets that separate the binary 'and'-operators from each other are redundant. So as long as we encounter the same operator (or an operator with the same priority) we leave out the brackets:

(A AND B AND C AND D) OR E

UIGLF is developed for readability only. It is not designed to work as an analysis format for computerised detailed analysis but it can be helpful for the understanding of the meaning of a certain GLF-expression.

We will give an example of the difference between GLF and UIGLF by providing their SDF definitions of arithmetic expressions:

```
%% "Highest Priority"
%% Notice how GLF does not allow brackets while UIGLF does
GLF-Identifier             -> GLF-AriExpr
GLF-Literal                -> GLF-AriExpr
GLF-FigLit                 -> GLF-AriExpr

UI-Identifier              ->  UI-PrimExpr
UI-Literal                 ->  UI-PrimExpr
UI-FigLit                  ->  UI-PrimExpr
"(" UI-AriExpr ")"         ->  UI-PrimExpr

"neg"  "(" GLF-AriExpr ")" -> GLF-AriExpr

UI-CHR-MINUS UI-UnaryExpr  ->  UI-UnaryExpr

%% Chain functions:
UI-PrimExpr                ->  UI-UnaryExpr
UI-UnaryExpr               ->  UI-AriExpr

"add" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"sub" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"mul" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"div" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr
"pow" "(" GLF-AriExpr "," GLF-AriExpr ")" -> GLF-AriExpr

UI-AriExpr UI-CHR-PLUS  UI-AriExpr        ->  UI-AriExpr
UI-AriExpr UI-CHR-MINUS UI-AriExpr        ->  UI-AriExpr
UI-AriExpr UI-CHR-MUL   UI-AriExpr        ->  UI-AriExpr
UI-AriExpr UI-CHR-DIV   UI-AriExpr        ->  UI-AriExpr
UI-AriExpr UI-CHR-POW   UI-AriExpr        ->  UI-AriExpr
```

It is hard to come up with an objective measurement for readability. What might appeal to one analyst could feel cumbersome to the other. One of the benefits of GLF and our modular approach is the fact that any kind of UIGLF can be connected to the system when that is preferred.

5. The Rewriters

The preceding chapters describe the "core issues" regarding the grammar of our various notations/languages. This chapter will focus on the disabbreviations, translations and transformations between these notations. We will frequently be showing pieces of the rewriters in the form of ASF equations; please keep in mind that these are only small parts of the equations and that they merely serve as examples.

5.1 Cleaning Up COBOL

Before we do anything with COBOL logical expressions we clean them up.

Disabbreviation

The most important issue in the cleaning up of COBOL expressions is removal of abbreviated expressions. When we traverse a logical expression from left to right we always take with us the last left (relational expression) operand and the last (relational expression) operator we encountered. Every time we encounter a relational expression with a missing operand or operator we fill in the gap.

We address this issue with the functions that take three arguments: a part of the expression that has to be cleaned, the last left operand we encountered and the last operator we encountered CleanUp(Expression_part, Last_LeftOp, Last_Op). We break up the expression into smaller pieces to eventually end up with relational expressions that have to be cleaned. Here are a few equations:

```
%% A complete relational expression, no cleaning necessary. Proceed with
new arguments
[CUR1a] CleanUpRel( LeftOp Op RightOp , Last_LeftOp , Last_Op )
                = ( LeftOp Op RightOp , LeftOp      ,      Op )

%% No Left Operand but already an operator, proceed with the new operator
[CUR2a] CleanUpRel(             Op RightOp , Last_LeftOp , Last_Op )
                = ( Last_LeftOp Op RightOp , Last_LeftOp ,      Op )

%% No Left Operand nor operator
[CUR3a] CleanUpRel(                      RightOp , Last_LeftOp , Last_Op )
                = ( Last_LeftOp Last_Op RightOp , Last_LeftOp , Last_Op )
```

Negations

Negations are handled as well (they are of course placed before the condition):

```
[CUR3b] CleanUpRel(                       ROp , Last_LOp , NOT Last_Op )
                = ( NOT Last_LOp Last_Op ROp , Last_LOp , NOT Last_Op )
```

Another issue is the removal of the negations that where within the condition. Consider the following rule that addresses the case in which

an expression like A < B OR NOT > C
must be converted to A < B OR NOT A > C
(continue with operand 'A ' and operator 'NOT >'):

```
[CUR2b] CleanUpRel(             NOT Op RightOp , Last_LeftOp , Last_Op )
                = ( NOT Last_LOp    Op RightOp , Last_LeftOp , NOT  Op )
```

The last two rules return logical expressions that begin with a NOT. These cases can produce redundant (double) negations. On a higher level, the Unary expression, this problem is solved (when we encounter a NOT before a Unary, we process that Unary first, if that generates a second NOT, we leave them both out):

```
%% Remove multiple NOT's
[CU3c] CleanUpUnary(     Unary1, Left_Op1 , Op1 )
                  = ( NOT Unary2, Left_OP2 , Op2 )
       ====================
       CleanUpUnary( NOT Unary1 , Left_Op1 , Op1 )
                  = (     Unary2 , Left_OP2 , Op2 )
```

5.2 COBOL to GLF

The COBOL to GLF translator is different from the COBOL Cleaner. The latter uses only one grammar (for both the source and the target language) while the former really translates one language to another, both specified in two different grammars in ASF+SDF.
So we are dealing with two grammars and a set of equations that translates terms of one grammar into their equivalents in the other grammar.
This can be seen when one looks at the modular structure of the translator. As a basis we have modules that are shared by both grammars (i.e. characters and digits), above that the two grammars are separated but with the same structure and in the middle we see the translation modules. It is illustrated by Figure 5.2.1.

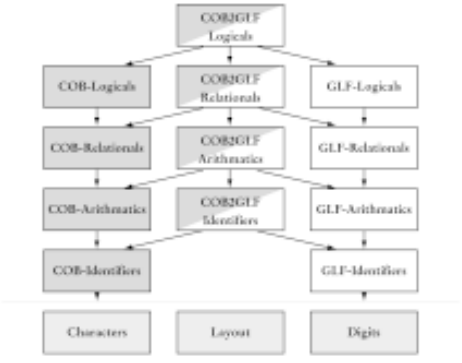


Figure 5.2.1

Since we are working with Clean COBOL the translating process is pretty straightforward. Here you see how a subtraction is translated (translate the left operand, translate the rest and subtract it):

```
[AE2] COB2GLF-MulExpr( COB-MulExpr1                    ) = GLF-AriExpr1 ,
      COB2GLF-AddExpr( COB-MulExpr2 COB-AddSubExprList ) = GLF-AriExpr2
      ====================
      COB2GLF-AddExpr( COB-MulExpr1 - COB-MulExpr2 COB-AddSubExprList)
                = sub( GLF-AriExpr1 , GLF-AriExpr2 )
```

Note the difference between the COBOL and the GLF sorts. Where COBOL distinguishes between several arithmetic expressions (such as Multiply-Expressions and Addition-Expressions) GLF only works with one sort, the Arithmetical-Expressions. Furthermore the COBOL expression works with two operands, an operator and the rest of the expression whereas GLF just has an operator on two Arithmetical expressions (containing the rest).

Logical expressions are handled similarly.

At the ground level of the translator several type conversions are needed. These include the adding of labels or the translation of typical COBOL language constructs in a more general way. These translations are not explicitly mentioned in the above since they are of merely technical use.

5.3 The GLF-Toolbox

Now that we have GLF we can manipulate it to make it meet our demands. As long as we stay within the GLF grammar we can perform all kinds of operations without modifying the other rewriters. We have implemented two normalizers and a few other operations.

The Conjunctive Normal Form CNF

Normal forms can be beneficial for both humans as well as computers in interpreting the value of a logical expression. We will describe the transforming of GLF expressions to their *conjunctive normal form*, or *CNF*.

A conjunctive normal form is defined like this:

- it is the conjunction (and'ing together) of clauses;
- each clause is a disjunction (or'ing together) of blocks; and
- each block is either an atomic expression or a negated atomic expression (preceded by a 'NOT').

In our case atomic expressions are relational expressions.

Translating this definition to GLF we have the following components:

- An atomic expression:
  in our case this is a relational expression such as 'eq(A, B)' (meaning A equals B).
- A negated atomic expression:
  this is simply a negated atomic relational expression like 'not(eq(A,B))'.
- Disjunctions of blocks:
  Just apply the disjunction to blocks for instance 'or(disjunctions_of_blocks , disjunctions_of_blocks)'.
- Conjunctions of clauses:
  i.e. and(conjunctions_of_clauses, conjunctions_of_clauses).

So if 'A', 'B' and 'C' are valid atomics.

and(A,or(A,or(B,not(C)))) = A AND (A OR B OR NOT C) is in CNF
and(A,or(A,not(or(B,C)))) = A AND (A OR NOT (B OR C)) is not in CNF

When working with the operators ‘AND’, ‘OR’ and ‘NOT’ three rules must be applied to get an expression in CNF:

moving the negations inwards (De Morgan):

not(A AND B) -> NOT (A) OR NOT (B)
not(A OR B) -> NOT (A) AND NOT (B)

In ASF+SDF it is written like this:

```
[mn3]
move-negs( not( and(GLF-LogExpr1, GLF-LogExpr2) ) ) =
 or( move-negs( not(GLF-LogExpr1) ) , move-negs( not(GLF-LogExpr2) ) )

[mn4]
move-negs( not(or(GLF-LogExpr1, GLF-LogExpr2)) ) =
 and( move-negs( not(GLF-LogExpr1) ), move-negs( not(GLF-LogExpr2) ) )
```

Now that all the negations are moved inwards we need to

distribute the ‘or’ over ‘and’:

or(A, and(B,C)) -> and(or(A,B) , or(A,C))

So the ‘and’s are moved to the outside and the ‘or’s to the inside.
The core of this is written down like this in ASF+SDF:

```
[do1] distribute-ors( or(GLF-LogExpr1, and(GLF-LogExpr2,GLF-LogExpr3))) =
and(
     distribute-ors(or(GLF-LogExpr1, GLF-LogExpr2)) ,
     distribute-ors(or(GLF-LogExpr1, GLF-LogExpr3))
)
```

When we encounter a disjunction with no direct conjunctions in it (this is because it is a default-equation, the former equation has got a higher priority and would match first in such a case) we distribute the disjunctions within the sub-expressions first. When they are transformed to CNF a conjunction could have been moved outside to one of the results (in this case the second operand of the conjunction was transformed to CNF, GLF-LogExpr4, and contains an conjunction). This means that the whole expression has to be processed again since this conjunction must be moved outside the whole expression as well.

```
[default-do6]
distribute-ors( GLF-LogExpr1 ) = GLF-LogExpr3, %% convert arguments
distribute-ors( GLF-LogExpr2 ) = GLF-LogExpr4, %% to CNF
is-and( GLF-LogExpr4 ) = YES %% this arg has one or more ‘and’s
==================
distribute-ors( or( GLF-LogExpr1, GLF-LogExpr2 ) ) =
distribute-ors( or( GLF-LogExpr3, GLF-LogExpr4 ) )
```

If we want to transform an expression to CNF we first move the negations inside (since that operation can generate additional conjunctions) and then distribute the disjunction:

```
[CNF1] move-negs      ( GLF-LogExpr1 ) = GLF-LogExpr2,
       distribute-ors ( GLF-LogExpr2 ) = GLF-LogExpr3
       ==================
       2CNF ( GLF-LogExpr1 ) = GLF-LogExpr3
```

In practice we found it useful to make a transformation that removes the negations while transforming an expression to CNF. When the negations are moved to the inside of the expression, i.e., up to the relational expressions we can make them disappear by inverting the relational operator:

not(A > B) can be written as (A <= B)
not(A = B) can be written as (A <> B)

or in GLF:

not(gt(A,B)) = lte(A,B)
not(eq(A,B)) = neq(A,B)

This means we will remove the negations instead of just moving them:

```
[CNF1] remove-negs    ( GLF-LogExpr1 ) = GLF-LogExpr2,
       distribute-ors ( GLF-LogExpr2 ) = GLF-LogExpr3
       ====================
       2CNF ( GLF-LogExpr1 ) = GLF-LogExpr3
```

The Disjunctive Normal Form DNF

The *Disjunctive Normal Form*, or just *DNF*, is very similar to the CNF. The difference lies in the use of the logical operators. Where in the CNF we have conjunctions of disjunctions, we work with disjuncted conjunctions in the DNF. The transformation is very similar to the CNF transformation, we will not describe it here.

Removal of operators

When testing the system we found it useful to add a few other operations to the GLF Toolbox. These operations remove certain operators from logical expressions. The GLF Toolbox can remove negations (remove them inwards and invert the relational operator), disjunctions and conjunctions.

Disjunctions are removed using the following rule (De Morgan):
or(A,B) = not(and(not(A), not(B)))

Conjunctions can be removed as well (De Morgan):
and(A,B) = not(or(not(A), not(B)))

5.4 GLF to UIGLF

The most interesting thing about the GLF to User Interface GLF translator is its bracket-policy. As described before it introduces brackets when operators have a higher priority and when they are necessary to force priorities. Brackets are left out when the same binary operator is used a few times successively, since in that case it does not contribute to the readability.

In GLF it is not very hard to see which sub-expressions must be resolved first (may need brackets). Whenever we encounter any kind of operator with two operands we know those two operands have a higher priority to be resolved and thus may be parenthesized.

Too many parentheses do not improve readability so we do apply a few extra rules:

- When an operator of the same priority is encountered don't use brackets.
  ((A + B) + C) - D can simply be written without brackets: A + B + C - D
- A negation does not need (extra) brackets
  (not(A)) is written as not(A)

- Within relational expressions we do not add extra brackets
  ((A) > (B)) OR (C) will be written as (A > B) OR C

So when we translate an operator such as the addition we translate the operands with brackets except when the first operator is an operator with the same priority as the addition (UIGLF-AriExpr-b-add), such as subtraction or addition itself:

```
[UI-AE1] UIGLF-AriExpr ( add(GLF-AriExpr1,GLF-AriExpr2) )
       = UIGLF-AriExpr-b-add( GLF-AriExpr1 )
         +
         UIGLF-AriExpr-b-add( GLF-AriExpr2 )
```

The function that places the brackets (UIGLF-AriExpr-b-add) is implemented to continue (without placing brackets) when it encounters an addition or substraction, and to continue while placing brackets (UIGLF-AriExpr-b) in all other cases:

```
%% With addition proceed without brackets
[UI-AEb-add1] UIGLF-AriExpr-b-add ( add(GLF-AriExpr1,GLF-AriExpr2) )
            = UIGLF-AriExpr       ( add(GLF-AriExpr1,GLF-AriExpr2) )

%% With subtraction proceed without brackets
[UI-AEb-add1] UIGLF-AriExpr-b-add ( sub(GLF-AriExpr1,GLF-AriExpr2) )
            = UIGLF-AriExpr       ( sub(GLF-AriExpr1,GLF-AriExpr2) )

%% In all other cases just add brackets:
[default] UIGLF-AriExpr-b-add( GLF-AriExpr )
        = UIGLF-AriExpr-b    ( GLF-AriExpr )
```

The `UIGLF-AriExpr-b` function just adds brackets around any expression.

With logical expression the same rules apply except in this case adding brackets means adding brackets except when a negation is encountered:

```
%% No, we don't add brackets here:
[UI-LEb3] UIGLF-LogExpr-b( not( GLF-LogExpr ) )
        = UIGLF-LogExpr  ( not(GLF-LogExpr) )

%% In all other cases we do add brackets:
[default]   UIGLF-LogExpr-b( GLF-LogExpr )
        = ( UIGLF-LogExpr  ( GLF-LogExpr ) )
```

5.5 Examples

To illustrate the use of the system we present some examples of rewritings.

A Small expression

We will be working with the following input in COBOL (unrestricted):

```
A > B OR NOT C AND D
```

When disabbreviated to Clean COBOL it looks like this:
(notice how the 'NOT' does not propagate since it does not preceed an operator)

```
A > B OR NOT A > C AND A > D
```

translated to GLF we get:
(notice how the conjunction binds stronger):

```
or(
  gt(id(vn(A)),id(vn(B))),
  and(
    not(gt(id(vn(A)),id(vn(C)))),
    gt(id(vn(A)),id(vn(D)))
  )
)
```

to UIGLF:

```
(A > B) or ( not(A > C) and (A > D) )
```

to CNF:

```
( (A > B) or not(A > C) ) and ( (A > B) or (A > D) )
```

to DNF (notice that the expression was already written in DNF):

```
(A > B) or ( not(A > C) and (A > D) )
```

to DNF without negations:

```
(A > B) or ( (A <= C) and (A > D) )
```

A large expression

The expression in (unrestricted) COBOL:

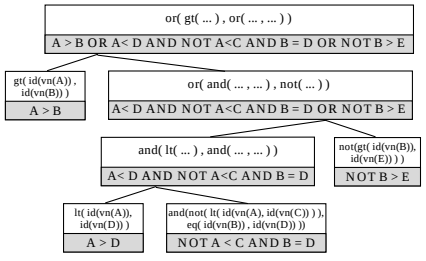
```
A > B OR < D AND NOT C AND B = D OR B NOT > E
```

After a cleanup we get:

```
A > B OR A < D AND NOT A < C AND B = D OR NOT B > E
```

It can be translated to the following GLF expression (pretty-printed by hand):

```
or(
  gt( id(vn(A)) , id(vn(B)) ),
  or(
    and(
      lt( id(vn(A)) , id(vn(D)) ),
      and(
        not( lt( id(vn(A)) , id(vn(C)) ) ),
             eq( id(vn(B)) , id(vn(D)) )
      )
    ),
    not( gt( id(vn(B)) , id(vn(E)) ) )
  )
)
```



Partial parse tree of 'A > B OR < D AND NOT C AND B = D OR B NOT > E' (after cleanup, before translation to GLF).

Notice how the disjunction (since it has a lower priority) is first in the GLF expression and therefore higher in the parse tree.

When translated to UIGLF we see how the expression becomes readable:

```
(A > B) or ( (A < D) and not(A < C) and (B == D) ) or not(B > E)
```

Notice that only when a new (logical) operator is encountered, brackets are used.

We can transfor m it to CNF:

```
( (A > B) or not(B > E) or    (A <  D) ) and
( (A > B) or not(B > E) or not(A <  C) ) and
( (A > B) or not(B > E) or    (B == D) )
```

and to CNF without the negations (for improved readability):

```
( (A > B) or (B <= E) or (A <  D )) and
( (A > B) or (B <= E) or (A >= C )) and
( (A > B) or (B <= E) or (B == D ))
```

to DNF (we notice that the expression was already in DNF):

```
(A > B) or
( (A < D) and not(A < C) and (B == D) ) or
not(B > E)
```

DNF without negations:

```
(A > B) or
( (A < D) and (A >= C) and (B == D) ) or
(B <= E)
```

6. Concluding Remarks after Part I

Now that we have presented our source (COBOL or Clean COBOL), our target (GLF or UIGLF) and our rewriters we will summarize the results.

6.1 Conclusions

In this part we have discussed the non-intuitive way logical expressions can be written in COBOL. We have presented a rewriting system that enables us to disabbreviate those expressions to a clean format from which we can translate them to a general logical format, GLF. The design of the rewriting system and the logical format is modular so the system as a whole can easily be extended to different languages.

We have shown that on GLF various translations and transformations are possible to improve the readability and analysis of GLF-expressions. This makes control (flow) analysis within legacy systems an easier task.

The "plugging in" of this system into an existing analysis system is described in part II of this thesis.

6.2 Future work

A lot of opportunities for extending the current rewriting system exist. The most eligible part to extend is the GLF toolbox. One can think of optimizations of logical expressions or statistics about the size, the amount of operators and whether an expression can be satisfied if one relational expression (within that expression) can not.

Other languages can be added to the system as well. Simply make a rewriter that translates Pascal logical expressions to GLF and the GLF-Toolbox as well as the UIGLF-translator will work on them.

A larger extension of the system would be the extension of GLF. The GLF described in this thesis can not handle function calls or other (user-defined) Boolean atomics.

Although lots of opportunities for future work are available we stress that the current system is enough to do some serious rewriting on COBOL (legacy) systems.

Part II

Embedding A Formal Specification in a Commercial Environment

The ASF+SDF Meta-Environment is an interactive programming environment for developing specifications in the algebraic specification formalism ASF+SDF. It provides a variety of tools useful for the development of stand-alone term rewriting systems based on this formalism.
The use of formal specifications can enhance the development of software especially in the area of software (re-, reverse-) engineering, but in practice it is used only rarely due to various practical issues.
This part of the master's thesis describes an industrial application of the ASF+SDF Meta-Environment especially focusing on porting it to, and integrating it in, an existing commercial system.
Though various applications of ASF+SDF in commercial environments are known [BDKM96] there has never been an integration on this level, where we take the functionality of the Meta-Environment and totally adjust it to its new surroundings.

7. Introduction

In this chapter we will briefly describe the two environments between which we are working. We will give global information on the properties of these environments and issues we must address during the embedding process.

A bridge, a way to connect two environments that are separated by water or gaps.

7.1 The Current Situation

Software renovation systems often consist of many components each with different functionality. During the renovation process various compilers or other translators/rewriters play an important role. In some of these phases a need for abstraction exists. Formal specifications of (programming) languages are a way to implement translators and rewriters in an abstract way.

Since the systems that need to be renovated often are of vital importance to society (such as systems at banks, airports, hospitals, government institutes or even military systems and nuclear plants) the renovation system must be reliable. A firm theoretical description of the software renovation system can be very beneficial to its reliability. An algebraic formalism like *ASF+SDF* supported by the *ASF+SDF Meta-Environment* [Kli92] is very suitable for these purposes. ASF [BHK89] and SDF [HKR89] together form a powerful way of describing specifications and term rewriting systems.

The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [Kli92] is a programming environment for writing ASF+SDF specifications. From a modular language definition and equations of a language it can parse, rewrite and pretty-print terms (programs) over that language. It consists of a variety of components together forming a programming environment.

With the development of the new ASF+SDF Meta-Environment [BKM97] a number of features have been added that open opportunities for integrating its rewriting capabilities in other systems. Existing (software renovation-) systems can thereby take advantage of the ASF+SDF specification formalism in some of their components without having to modify existing ones.

Still it is not just a matter of ‘plugging in’ a rewriting system into an other system.

Problem

Despite the functional benefits of formal specifications they are far from widely spread outside the scientific world. There are few complete solutions (like the Meta-Environment) that provide functionality for formal specifications and they are mostly developed

in scientific environments (like universities) rather than in commercial (like corporations) environments. Several differences between a scientific and a commercial environment exist that have led to this unilateral development.

The development of such a system requires a lot of theoretical background and research in order to give it its necessary firm basis. It is a long-term project that can take years to finish. Companies often do not take the risk to invest their money, time and personnel into such a time consuming project, especially in the fast IT business. Even purchasing a system like that, if a complete system is found, is not straightforward. Companies have different requirements and digital infrastructure than most scientific institutes.

Scientific institutes are willing to take chances and time to develop such a project because their interest lies in the research as well as the results. They are also willing to share and exchange knowledge with other institutes rather than trying to keep the competition ignorant of their projects. The ASF+SDF Meta-Environment is no exception and is also developed in a scientific environment. It has already been used in some commercial projects [BDKM96] but it has never truly been embedded in an external system.

Our question, our goal

We are going to describe the process of a low level integration of the ASF+SDF formalism using the functionality of the Meta-Environment in a commercial environment. This means that the formalism will be a part of a commercial *application* rather than part of a *process*. We will show how it is done, which issues have to be considered and which choices have to be made. This project can function as a pilot or an example for future projects.

The main questions of this part of the thesis are:

- Can we embed the ASF+SDF functionality of the Meta-Environment in a commercial environment?
- Which problems do we encounter?
- Which choices have to be made?

Our goal will be to extract only the functionality we need from the Meta-Environment, to modify it in such a way that we can make a stand-alone component of it and to embed it in an external system.

Lots of problems need to be investigated and solved in order to make the low level integration possible. We encounter system design differences, platform differences and language differences that have to be solved.

7.2 Environment Differences

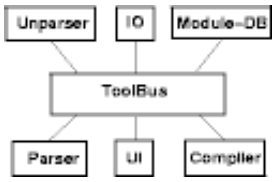
When trying to integrate the Meta-Environment in a commercial environment problems on different levels arise. Of course we will encounter various technical incompatibilities, but also on a higher level several issues regarding the design of both systems need to be addressed.

System Differences

We could try to make the whole environment part of the system but it is preferable to only use the functionality we need and build this into a tool or some kind of component that can be ‘plugged’ into the existing system. Such a component should be set up in a way that it is as independent as possible of both the renovation-system and the Meta-Environment. It should be ‘unpluggable’ at any time without messing up the other sys-

tems. We would like to have as little user-interference as possible so it can stay a clean 'black box' with a clear input and output.

We would also prefer to leave the exact ASF+SDF specification (the term rewriting system, *TRS*) as a separate part within our component so its rewriting capabilities can easily be changed without having to modify all the other, technical parts, of the system.



In the Meta-Environment the components are connected by the Toolbus.

The Meta-Environment exists of a large collection of tools that communicate with each other using the Toolbus [BK98]. The tools are independent executable applications. Since we want to embed their functionality in an external system they should be translated into libraries. This change in design means we have to replace the Toolbus with a new tool that explicitly handles the communication between the other tools.

Platform and Language differences

When we try to embed a 'scientific system' like the Meta-Environment in a 'commercial system' we run into the differences between scientific and commercial computer platforms.

Sun's Solaris is the development platform on which most of the Meta-Environment is developed.

The Meta-Environment is developed on the Unix platform (i.e. Sun Solaris, Linux, Apple Mac OS X, Compaq Tru64 etc) while in commercial corporations the most widely used development computer-platform is Wintel (Windows on Intel-processor based machines). Fortunately, in the Unix world it is very common to distribute scientific applications and tools with their source-code (and makefiles). This makes it more important to Unix-programmers to program platform independent (at least Unix-platform independent) and make relatively 'easy-to-compile' distributions of their sources.

Windows NT has got quite a bit of Unix or Unix-like core, so porting the code seems at least possible.

Still lots of differences arise between the platforms. Like it or not, source code on a Unix machine compiled with compiler A can have different behavior from the same source code compiled with compiler B on a Wintel machine, if it even compiles. Functionality that is available on the Unix platform might not be available within Windows NT; some functions have slightly different side effects while others have different preconditions and so forth.

When we have the ideal situation of a complete system under Wintel we discover a new challenge: language differences. All the source-code we need from the Meta-Environment is written in ANSI C [KR88]. C is a very common programming language that is used in a lot of commercial environments. However larger systems like the one in which we like to adopt our rewriting system often are written in more than one programming language. In our case the core of the external system is mostly written in Java [Eck98] whereas the graphical user interface has been made using Visual Basic [SM97].

Microsoft's Windows NT is the development platform used by TriLoc to develop most of their tools.

Our goal will be to filter out any vital platform incompatibilities and develop an interface/adapter between our component and the existing system, to overcome the language differences.

8. The Components

In this chapter we will give an in depth description of our existing 'source' components and the component we are going to build, the 'target'.

To get a clear view on the whole picture it is necessary to understand its components.

8.1 An in-depth look at the Meta-Environment

The Meta-Environment is a complex system consisting of many different tools that work together. In order to adapt this system to its new surroundings we have to have detailed knowledge about its technical background and its design. The following sections discuss the parts of the system that are important to our project.

ATerms and AsFix

In order to exchange data between its components the Meta-Environment uses *Annotated Terms* [BJKO99]. Annotated terms, or just *ATerms*, form a data structure that is ideal for communication between tools because they are platform and language independent and simple to create, compact and extend.

To make ATerms available to other applications a C-library [JO99] has been developed that provides high performance operations on ATerms as well as input-output and efficient memory management. Input-output is possible in textual or in a more compressed binary format. A minimal amount of memory is required when working with ATerms thanks to the technique of *maximal sharing*: only new terms are created, if a term to be constructed already exists, that term is reused and only a reference to that term is created. Memory management is done by a built-in mark-sweep garbage collector. Although a lot of redundant terms are produced during the rewriting process, the use of maximal sharing gives the ATerm library a sometimes spectacularly efficient memory usage, as described in [BKO99]. This is beneficial since ATerms can also be used as an internal data-structure for data (such as parse trees) within running applications. In order to make use of this feature the components have to share the same 'ATerm environment', with the same stack, the same garbage collector and the same memory manager.

To represent ASF+SDF specifications and terms a special incarnation of ATerms is used, the ASF+SDF Fixed format, or just AsFix. AsFix terms can be regarded as parse trees of ASF+SDF specifications. They can be managed just like ATerms but are easier to use when working with ASF+SDF. An extra C-library is available, the support-library, to give support for creating and manipulating AsFix terms.

We will mostly be working with AsFix but since every AsFix term is a valid ATerm as well, they can be regarded ATerms.

The Compiler

The compiler perhaps is the most important part of the system for our project. It compiles the ASF+SDF specification (represented in AsFix) to ANSI C code. Since we want to do our rewriting without an interactive environment and on a different platform we prefer to use C code, that can be compiled to a library and used by our external systems, to represent our TRS rather than the ASF+SDF specification itself. So without the compiler, integration to the level we prefer would surely be impossible.

More information on the ASF+SDF to C compiler can be found in [BKV98].

The SGLR-Parser

Dr. Eelco Visser.
Expert on sglr-parsing

Another important component in the ASF+SDF Meta-Environment is the parser. Before we can rewrite our input it needs to be recognized and identified, that is where a parser comes in. The parser should support all kinds of grammars that can be specified in the Meta-Environment with SDF. It needs to be language independent and capable of parsing SDF-specifications. The parser that is used is the *Scannerless Generalized Left-to-right Rightmost derivation parser*, or just *SGLR-parser* [Vis97]. In many cases *scanners* are used to divide the input into lexical tokens (specified by the lexical syntax, the regular grammar) which are in turn parsed by a *parser* who returns a tree according to the context-free syntax. A downside to this approach is the fact that no knowledge of the parsing context is available to the scanner meaning that the context of a token can not be used to disambiguate its meaning.

The 'S' in SGLR stands for 'Scannerless' this means (among other things) that the *lexical* and *context-free* syntax are both handled by a single context-free analysis phase thereby providing the ability to take the context of a token into account during the parsing process. For further disambiguation *reject-rules* and *context-free* restrictions can be used providing ways to, for instance, force 'prefer longest match' and 'prefer keywords over tokens'.

The 'G' is taken from 'Generalized'. Generalized-LR parsing [Vis97-2] basically comes down to the fact that when an ambiguity is found the parser produces all possible parse trees of the input, resulting in a so-called *parse forest*. In practice we found the forests very helpful while disambiguating our grammars, it proved to be a nice debugging facility.

The 'L' stands for left-to-right scanning of input.

And finally the 'R' stands for rightmost derivation in reverse. The consequences of these properties are beyond the scope of this thesis.

Creating a Term Rewriting System using the Meta-Environment

The porting process reaches beyond the need to just compile the code provided by the compiler on a different platform. Translations are needed as well as choices regarding design.

To clarify our needs we will now describe how a 'stand-alone' rewriting system (that is operational under Unix) is created with the Meta-Environment. The process is illustrated by Figure 8.1.1.

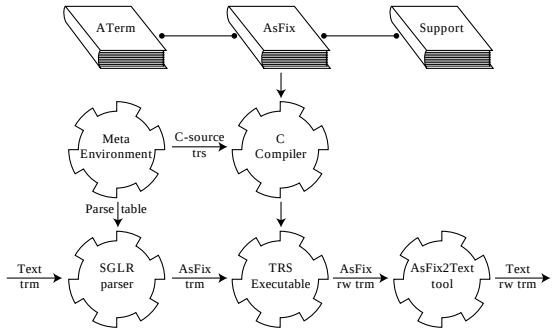


Figure 8.1.1

- The built-in compiler of the Meta-Environment translates our specification into ANSI-C.
- The obtained C-source is fed to a C-compiler along with the ATerm and AsFix libraries as well as an extra *support library*, which can then be compiled to an executable application, the TRS executable.
- This term rewriting system executable cannot just rewrite strings of terms; it requires a parsed term in AsFix format as input. In order to obtain a parsed AsFix-term from our input string we need to parse it and embed the production rules of the input (gathered from the signature of our ASF+SDF specification) into that term.
- A textual representation of an input term can be parsed to an AsFix-term by the SGLR-parser given some parse table in ATerm format.
- This AsFix term can be used as input for our TRS-executable which will produce a rewritten term.
- This rewritten AsFix-term can be translated to text in various ways with some kind of AsFix2Text tool (functionality is provided by the AsFix library).

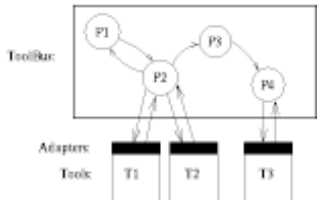
With a design like this we can, e.g., create an ASF+SDF specification for the Booleans, feed the term ‘true | false’ to SGLR, rewrite it and receive ‘true’ from our AsFix2Text tool.

8.2 A TRS component for Windows NT

Now that we know the sequence of steps to rewrite terms with an independent TRS we need to make decisions about which parts of the system we need to port, how we will make them work together and in what way they should communicate with the external system.

What can we port?

We will leave the Meta-Environment and the Toolbus untouched since these systems are very complex, still partly under construction, hard to port and therefore beyond the scope of this project. Besides it is our goal to create a library rather than a set of tools that we have to embed in an existing system. This means we will still have to create the source code of our TRS on Unix but we can rewrite terms on another platform. Perhaps when this project succeeds, and most of the tools and libraries are ported, it will be interesting to look at possibilities to port larger components like the Toolbus to Windows NT as well.



The Toolbus is a software application that utilizes a scripting language based on process algebra to describe communication between software tools.

Without the Toolbus it is necessary to think about a way the remaining tools have to communicate. In our case we need to connect the TRS to a larger system and the ideal situation is an independent stand-alone library that can be used in the current system. This means we have to modify the tools to libraries with functionality independent of the Toolbus and other unused components. Another thing we have to keep in mind is that we want the system to be modular with at least the TRS as a separate part so we do not have to modify the whole system when we use a different ASF+SDF specification.

Thanks to the Unix-way of installing software we have the source-code of every component we need so we can modify it in a way to make it suitable for our purposes. We prefer to leave all automatically generated code of the TRS untouched, because when we modify that we will have to do so each time we create a new TRS.

The complete system will look like figure 8.2.1.

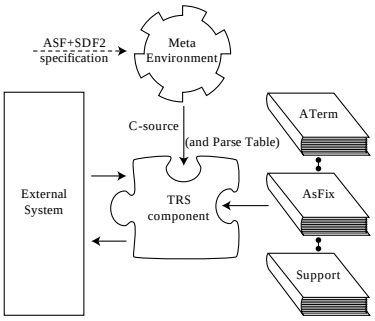


Figure 8.2.1

The Meta-Environment is still operating under Unix while the rest of the system is ported to Windows NT. The TRS-component that must be created can be a library, a stand-alone tool or something in between like a dynamic linked library that can be loaded and called from the external system.

A more technical thing we have to keep in mind is the use of the ATerm library. A lot of the components we need, make use of this library and if we want to fully exploit its maximal sharing capabilities, we must try to make them use the same ‘ATerm-environment’ (instance of the ATerm library). Initializing and using the library explicitly for each component would make the system very inefficient.

Interfacing our TRS with the existing system

As mentioned above we have several choices regarding the functionality of our TRS. We can choose to leave this tool as small as possible thereby requiring the external system to translate the in- and output to and from AsFix. We can also choose to integrate the translations within the TRS and hide the AsFix notation from the external system.

We will discuss three possible interfaces in some detail.

Interface 1: The Indirect Interface

The indirect interface leaves the external system unaware of the AsFix. A textual representation of a term (like ‘true | false’) can be fed to the TRS and it will produce the rewritten term in a textual representation (like ‘true’) as output (Figure 8.2.2).

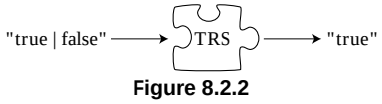


Figure 8.2.2

With this interface the TRS-component consists of 4 subcomponents: an adapter to handle communication and flow of control, an SGLR-parser subcomponent, an AsFix-to-text converter subcomponent and of course a TRS subcomponent (Figure 8.2.3).

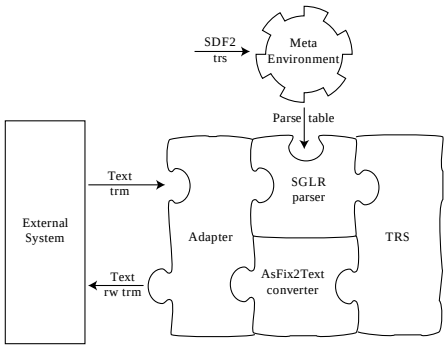


Figure 8.2.3

A benefit of this implementation is the independence between the component and the external system. Terms are being rewritten and there is no need for further interaction or tuning between both systems. A totally different TRS-component can be linked to the system without any necessary modifications.

Performance, however, can become a problem with this implementation. In the external system the input is produced in some way and has probably already been parsed. Now it needs to be translated back to text which is being parsed again by our component, rewritten and converted back to text (AsFix2Text) to be parsed (for the third time) by the external system.

Interface 2: The Direct Interface.

The direct interface leaves the external system with the task of creating AsFix terms. This means that the system must have access to the ATerm library (and preferably the AsFix library as well) and must have some knowledge about the signature of the ASF+SDF-specification which must be embedded in the (parsed) input term. The TRS just rewrites these terms and returns them in AsFix format. The ATerms can be transferred in the compact binary format (Figure 8.2.4).

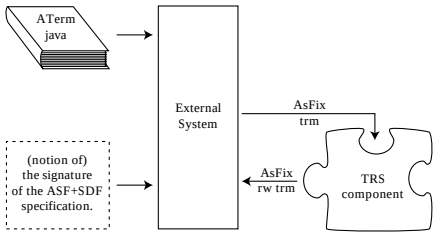


Figure 8.2.4

The parsing is done in the external system which means it can be dedicated and optimized for the given TRS. No additional translations from parsed-terms to text, or vice versa is needed and with appropriate implementation some time is saved.

Of course there can be no independence between the external system and the TRS since parsing has to be done according to the signature of the TRS. The external system has to have knowledge of the production rules of the TRS and the ATerm library. If another TRS is needed this means that extensive changes are needed within the external system in order to make communication possible.

Interface 3: Total Integration

The third possibility is total integration. The key feature of this implementation is that the external system has the same 'ATerm environment' as the TRS. This is the solution for real maximal sharing. Providing us with a very memory- and time-efficient solution. To make this possible we need to have as much ATerm functionality available within our external system as possible. Unfortunately such functionality is not yet available in Java, and language differences, especially the memory management differences between Java and C, make maximal sharing almost impossible. This implementation also puts a lot of constraints on the external system making it totally dedicated to a specific TRS.

Our choice

In our situation the independence between the components is of vital importance. The Meta-Environment, the external system and the TRS are all under construction and we must be able to 'plug' and 'unplug' a specification at any desired moment. In certain situations it is even desirable to use different TRS's from the external system. This leads us to the first type of interface.

Furthermore the construction of AsFix input terms in the external system is hard to do, since it also means that the signature of the ASF+SDF specification should be parsed (or the parse table from the Meta-Environment should be interpreted).

8.3 Wrapping up the design

The idea is to write code that uses the functionality of our three subcomponents (SGLR, the TRS and AsFix2Text) in order to achieve the desired effect. We could also have chosen to use these subcomponents as separate tools (and thereby modifying the existing code as little as possible) but that would mean we had to initialize and instantiate a new instance of the ATerm library for each subcomponent. Now we only instantiate the ATerm library once meaning we have minimal overhead and maximal sharing. The code is stored in a library called the *Meta-Adapter*.

The Meta-Adapter can be viewed upon as a kind of sequencer. The components under Unix are mostly tools (stand-alone applications) that communicate with each other using the Toolbus. We want to modify these tools in such a way that they become libraries, independent of the Toolbus. This mostly requires the removal of code (such as main-functions and Toolbus functionality).

In our approach the Meta-adapter takes over the communication between the subcomponents previously handled by the Toolbus.

An in depth look at the Meta-adapter

The Meta-Adapter is the most interesting part of the Windows NT system when it comes to understanding the design. We will now give a more detailed description of its behavior by focussing on some of its source code.

Along with some error handling functions the Meta-Adapter contains an initialization function in which the parse table is loaded (parsed) and a rewrite function that starts the actual rewriting process.

Initialize TRS

The initialization function gets the location of the parse table and tries to load it. When the parse table is parsed successfully it initializes the rewriter.

The parsing of the parse table is the most time-consuming process within the rewriter so it is very beneficial for the efficiency to parse it only once.

In pseudo-C it looks like this:

```
/* Pseudo code of Metadap.c: the implementation of the
 * indirect interface.
 * The code is not clean C, it represents only a fraction of the
 * real code but it is more readable leaving out unnecessary
 * details.
 */
void initialize_trs(char * ptable) {
     AFinit(...);          /* initialize the ATerm library      */
     read_parsetable( ptable );
     init_rewriter();      /* initialize the TRS-subcomponent   */
}
```

The initialization of the ATerm library through our whole component only occurs in this function. Since initialize_trs is called only once this means that all our ATerm-operations that are initiated by the rewriter, parser or other subcomponents will operate on the same ATerm environment. This ensures us of maximal sharing within our system to keep our memory usage as efficient as possible (by the current ATerm library standards).

Rewrite Term

Rewriting terms is the actual goal of our system. Once the initialize function has been called the rewriter will be executed many times.

In pseudo-C it looks like this:

```
char *rewrite_term(char *input_term, char *ptable)
{
     ATerm at;
     char * output_term;
     at = SG_Parse(SG_LookupParseTable(ptable), input_term);
     at = innermost(at);           /* Rewrite     */
     AFsource(at, output_term);    /* AsFix2Text  */
     return output_term;
}
```

As seen above the rewrite_term function gets an input term in string format and returns the rewritten term (also in string format).

The subcomponents are not called as separate tools (with a script or something) but a more low level approach has been taken by directly calling certain functions that exist within the sources of the components. This is a major deviation from the Toolbus approach that is normally used within the Meta-Environment. The subcomponents are linked as static libraries.

Improvements of the Meta-Adapter

For the future it might be interesting to link the TRS-sub-component dynamically allowing the use of other TRS's without having to recompile the entire Meta-Adapter. This would mean that we would have to re-initialize the new rewriter and perhaps close (de-initialize) the old one. A user can locate TRS-dll's and their accompanying parse tables manually or through some repository. We, however, regard this extension as a task that lies beyond the scope of this thesis and leave it for future work.

Combining the Sources

The Meta-Adapter has external references to functions that exist in the TRS-subcomponent (the generated TRS-code) the latter is linked along with the Meta-Adapter to a higher component which does the actual linking of the whole component.

This design enables us to rewrite terms in a clean, independent and an efficient way. The TRS-component can be replaced by any other rewriting mechanism without the

external system being modified in any way whatsoever, its functionality is well defined (when the ASF+SDF specification is known) and clear and it makes use of the efficient ATerm memory management.

The Compilers

We deal with a system that needs to be ported from Unix to Windows NT. An ANSI C compiler (like CC or GCC) is used under Unix and we use Microsoft's Visual C++ compiler under Windows NT.

9. Solving Differences

Now that we have defined our current situation and our desired result, it is time to actually implement it. We will be discussing many of the global differences between platforms and languages that have influenced our design strategies or are otherwise worth mentioning. Of course we will also present ways to solve the problems.

Platforms differences seem to reach further than just technical issues, as shown by the 'Operating System Sucks-Rules-O-Meter' which expresses opinions found on websites on the internet.

9.1 Binary IO

When working with binary IO we ran into a problem the compiler did not warn us for. Within Unix there does not seem to be a real difference between text and binary file IO. When using Windows it is very important to tell the system you are doing your IO in binary mode. If you forget to do so, strange things will happen. We, for instance, were confronted with the fact that the hexadecimal pattern '1A' would simply make the program crash as soon as it was encountered during the IO. When this pattern was read from a file that was open for non-binary reading, it was interpreted as an end-of-file causing the system to stop reading. Of course this led to a variety of errors that where hard to trace.

The ATerm library was doing its binary IO correctly when it came to files, but when stdin and stdout were used as files, the library forgot to actually set them in the binary mode. Within Unix this does not result in problems so we were the first to interpret this as an error. Along with the ATerm-library developers we decided to fix this problem within the binary file-IO function of the ATerm-library with the windows specific _setmode statement:

```
#ifdef WIN32
if(_setmode(_fileno( file), O_BINARY ) == -1) {
     perror("Warning: Cannot set file to binary mode.");
}
#endif
```

By solving this problems within the code of the ATerm-library we make sure all the code that makes use of the functions is corrected. It makes the library both easier and safer to use.

Memory Management: Getting the registers.

Of course low-level memory management brings lots of problems when porting code. We encountered various incompatibilities. We even discovered a bug in a memory initialise statement during the porting process. A certain part of memory had to be initialized to zero but only part of it actually was set. The function memset was called which sets a specified number of bytes to a certain value (in this case zero) unfortunately it was called to set a number of ATerms to zero without providing it with the size of an ATerm: the call was `memset(start-pointer, 0, nr-of-ATerms)` but it should have been `memset(start-`

`pointer, 0, nr-of-ATerms * sizeof(ATerm) )`. Within Unix this did not result into crashes because memory seems to be zero by default; within Windows NT however the bug started to become a problem because memory is explicitly initialized to a different value and the library started to crash at unpredictable moments.

The garbage collector within the ATerm library uses the mark-sweep technique [BJKO99]. In the mark-phase the memory must be checked for ATerms in order to mark them. This includes checking the stack and the registers. In Unix the sigsetjmp-function places the registers in a data structure which is (automatically) placed on the stack. The stack can then be checked from top to bottom. Unfortunately this functionality is not provided under Windows. We need our own way to check the registers. On the internet we found the Boehm garbage collector [Boe99], which is available for both Unix and Windows, from which we concluded that the only way to solve our problem was to use assembler. The assembler code copies the values of the registers into local variables which are explicitly being fed to the garbage collector. Illustrated by the following code (this is not the exact code, it merely serves as an extra explanation):

Hans J. Boehm. Creator of the Boehm garbage collector.

```
__asm {            /* Get the registers into local variables
     mov r_eax, eax  * to check them for ATerms later. */
     mov r_ebx, ebx
     mov r_ecx, ecx
     ...
}
reg[0] = (ATerm) r_eax; /* Put the register-values into an array */
reg[1] = (ATerm) r_ebx;
reg[2] = (ATerm) r_ecx;
...
/* First traverse the reg-array
 * to count the nr of ATerms that were in registers
 */
  for(i=0; i<NR_OF_REGISTERS; i++) {
    if (AT_isValidTerm(reg[i]))
      AT_markTerm(reg[i]);
    if (AT_isValidSymbol((Symbol)reg[i]))
      AT_markSymbol((Symbol)reg[i]);
  }
```

9.2 Language Differences

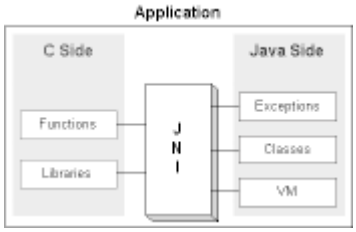
The commercial environment in which we have to adapt our system is written in Java. To make communication possible between the two systems an interface between C and Java is needed. This interface is far from trivial since Java and C are different languages; not just when looking at the object oriented way Java is designed, but also by means of implementation. Java for instance has a built-in garbage collector that keeps track of everything within the Java-program that uses memory. Fortunately, with Java the *Java Native Interface* or *JNI* [JNI99] is developed to make communication with C possible.

The JNI provides a set of C functions that take care of various translations and memory incompatibilities. It can take a Java class-file with definitions of native methods (which are the Java-counterparts of C-functions) and generate a C header-file with the definitions of the C-functions.

A Java native method, with its C-counterpart:

```
   public static native String
      DoItInC(String istr, String ptable, String ssym);

   JNIEXPORT jstring JNICALL
      Java_DoItInC (JNIEnv *, jclass, jstring, jstring,
jstring);
```



The Java Native Interface, talking C and drinking Coffee.

These definitions must be used as definitions of the actual C-functions. In C several issues regarding Java's memory management and different types have to be addressed. For instance strings have to be decoded before they can be used in C and simply freeing memory of variables declared in Java is not allowed.

```
/* Decode a javastring */
*c_string   = (*env)->GetStringUTFChars(env, j_string, 0);

/* Release a string so the Java garbage collector can remove it */
(*env)->ReleaseStringUTFChars(env, j_string, c_string);
```

When these issues have been addressed correctly C-functions can be called from Java, arguments can be passed to them and return values can be received from them (the JNI provides even more functionality, but we will not be using that in this project). The JNI within our system is shown in Figure 9.2.1.

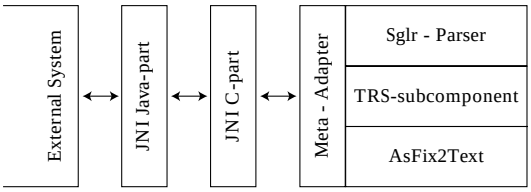


Figure 9.2.1

The external Java system communicates with a Java package which has a C-counterpart with which it communicates using the JNI. The JNI C-part communicates with the Meta-adapter which in turn is connected to the subcomponents of our TRS-component.

Error Handling: No Crashes Please.

At various points in the rewriting process in the TRS-component things can go wrong: unparsable input-terms can cause the system to crash, illegal parse tables, a TRS-subcomponent with a (slightly) different signature than the parse table etc. An *error* does not have to be a problem, a *crash* is.

Our system is embedded in a larger Java system that requires safe and secure error-handling. We do not want the whole system to crash just because our TRS-component has encountered an error. Fortunately, the ATerm library provides a way to implement custom error-functions. With the ATsetErrorHandler-function we can pass our own error-handler to the ATerm-library. An error message will be passed to this handler with optional arguments.

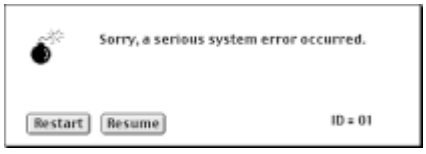
```
void ATsetErrorHandler(void (*handler) (const char *format,va_list args))
```

When we encounter an error it is important to stop the current processes to prevent more errors, or even crashes, from occurring. We must return to a safe state in which we can report an error to the external system. In Java, a try-catch statement can be used to this. In C we must do it 'by hand' with the functions setjmp and longjmp:

```
if(setjmp(env) == 0) {
     /* do something,
      * if something goes wrong call longjmp(env,-1)
      */
} else {
     /* Longjmp has been called: an error has occurred */
}
```

When we encounter the call to setjmp for the first time it returns zero and we continue with the code. Within our custom error-handler, that is called every time an error occurs, we have a call to `longjmp` and when that is called the stack (pointer) is changed to the value it had when `setjmp` was called, i.e. we travel back in time to the `setjmp` call. This time however `setjmp` returns -1 and instead of performing the statements that caused the error again we proceed with the else-block.

Throwing errors from C to Java is possible (with JNI) but in our case we did not find it preferable. We wanted to have as little Java-C-communication as possible. We decided that on an error the TRS-component just returns an empty string and that the Java adapter can check afterwards whether an error has occurred or not. If one has occurred it can then explicitly get the error message and throw an exception if that is desired. This way we only call C-functionality from Java and not the other way around, meaning that the external system really is an independent tool with no links to the external system. In Java the code looks like this:




```
public String RewriteTerm (String input_str) throws TrsException {
  String output_str = DoItInC(input_str);
  if (CErrorTest() == 0) {
    return output_str;
  } else {
    throw new TrsException( GetCErrorMsg() );
  }
}
```

The `DoItInC` method (which is a JNI-call to a C function) always returns a valid string (even on an error). The `CErrorTest` method checks whether an error has occurred and if so, the `GetCErrorMsg` method gets the error message that can then be thrown as a Java-exception. This way the external system does not even know the TRS-component is written in a different language.

10. Concluding Remarks after Part II

After this porting and embedding process one should be able to start and run a TRS under Windows

Adding it all up, some things can be said about our and other approaches on embedding ASF+SDF functionality in commercial environments. We will also be describing a few practical / technical issues.

10.1 Practical Issues

During the porting and testing process we discovered a few incompatibilities that still need to be addressed.

Sun and Microsoft have different virtual machines

As described earlier the TRS component in Windows throws an exception when an error is encountered. Strange enough this still resulted in a crash when it was tested. After reviewing the code and the input, we decided to use a different virtual machine.

In our situation at TriLoc software engineering we are working with an external system that contains a lot of Java and has an interface which is created with Microsoft's Visual Basic. In order to make the communication between the interface and Java as smooth as possible we used Microsoft's Java virtual machine. This generated crashes when we were throwing exceptions. Sun's virtual machine (for Windows) did not have this problem. We hope this issue is addressed in future versions of Microsoft's virtual machine.

Generated C-code and Old Meta-Environment React Differently

At the time of this writing the development of ASF+SDF specifications was still done using the old Meta-Environment (based on the Centaur system).

We noticed that in some cases, some terms where not rewritten to their normal form under Windows when under the Meta-Environment they were. This was the fact because the generated C-code of the ASF+SDF specification (generated by the Meta-Environment) is more specific about the use of types than the old Meta-Environment.

The issue can be looked upon as a type casting problem. In the old Meta-Environment subsorts of sort A are implicitly casted to work on rewrite rules of A. The generated C-code expects the ASF+SDF programmer to be more specific about his or her type usage.

This is a known problem and will either be solved or correctly documented in the future by the Meta-Environment development team.

Generated C-code will not compile

In rare cases the generated C-code simply will not compile because of a type incompatibility. Since the developers of the compiler within the Meta-Environment believe it will

be beneficial for the efficiency, they do as little explicit typecasting as possible. The case we discovered has been reported to them and we expect this issue to be resolved in the next distribution of the system.

Not ALL C code is Generated by the Meta Environment

Strictly the Meta-Environment does not generate all of the source code needed to create a TRS under Windows. An important file called init.c is not generated by the built-in compiler but is implicitly defined in the (generated) Makefile. So after the C-code is generated it is necessary to compile it (under Unix) to get the init.c file.

This is a known fact (but we are not sure it is a bug or a feature). At this time there is no information available on whether this issue will be addressed or not.

Memory Leak!

When we rewrite a term under Windows we have a memory leak that occurs when the term is parsed (and the parse table is looked up in the parse table database). This bug is known both at the Meta-Environment development team as well as TriLoc but it has not been precisely corrected or localized yet.

10.2 Conclusions

Embedding the functionality of a scientific system like the ASF+SDF Meta-Environment in a commercial environment is possible. To overcome system-, platform- and language differences various choices have been made. Some parts need to be rewritten, some parts have to be grouped together to acquire a system which is as safe as possible without losing the necessary structure.

We have described the integration using a prototype of the Meta-Environment (since it is still in a development stage), during the process we have given feedback to the developers which they have used to adjust the system to make future portings and projects like this much easier.

10.3 Future Work

In the future, when there is need for several term rewriting systems to work with an external system, dynamically linking the TRS subcomponent can be very beneficial. In the current situation the TRS-subcomponent is fixed to the system at compile time, while for instance the parse table can be chosen at run time.

Future work might also include the porting of the Toolbus and the Meta-Environment itself but we suggest to wait with this until the development of the latter has been finalized (at least to a certain degree). The TRS-functionality, the ATerm and AsFix libraries and the SGLR-parser can already be used in cases like this; thereby bridging the gap between this scientific and other commercial systems.

Concluding Remarks

11. Concluding Remarks

Returning to the Introduction of this thesis we look at which goals we have achieved.

11.1 Conclusion

Returning to the beginning of this thesis we will review what has been done.

1. We have described a COBOL rewriter in the algebraic specification formalism ASF+SDF that can be used to disabbreviate COBOL logical expressions. This was the initial assignment for this master thesis.
2. We have also defined a language independent logical format (the GLF) on which various rewritings are possible. These rewritings include transformation to normal forms, the removal of logical operators and the important translation to the readable UIGLF format. This makes the system much more powerful and opens a world of opportunities for future extensions.
3. We have ported major parts of the ASF+SDF Meta-Environment rewriting system (that supports ASF+SDF) to Windows NT. This allows future projects to make use of ATerms, AsFix, SGLR and the support library under NT. We solved some problems and gave feed back to the Meta-Environment developers for future releases.
4. We have created a stand-alone TRS component that can be linked to the TriLoc Logic Mining system (or any other system for that matter) whenever that is found beneficial. External systems need only minor adjustments and can function properly without the use of the TRS as well.

Because of practical problems that where beyond my reach I have not been able to do benchmarks on the system as it was implemented at TriLoc. For statistics on the performance of the Meta-Environment under Unix please refer to [BKO99]. The rewriting system has been tested on a ‘real life’ system from a Dutch government agency and it worked correctly without unacceptable performance problems. The system had several thousand lines of code containing several hundred conditional expressions that were rewritten to Clean COBOL, GLF and finally UIGLF in CNF without negations.

It is my personal opinion that the system can be commercially competitive and can be applied in many other areas where rewriting is done within the Logic Mining system (or other systems).

Although the *technical* and *theoretical* issues that we have described in this thesis are important we can not forget the *practical* side. I have had a great time working at TriLoc and staying in touch with the CWI and the UvA. This shows that commercial and scientific ‘flavors’ can ‘blend’ together in a successful project.

References

[Alto92] Reasoning Systems, Palo Alto, California. Refine User's Guide, 1992.

[BDKM96] M.G.J. van den Brand, A. van Deursen, P. Klint, A.S. Klusener and E.A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, Algebraic Methodology and Software Technology (AMAST '96), volume 1101 of Lecture Notes in Computer Science, pages 9-18. Springer-Verlag, 1996.

[Boe99] H. J. Boehm. Boehm Garbage Collector. Code and documentation can be found at http://reality.sgi.com/boehm_mti/gc.html

[BHK89] J.A. Bergstra, J. Heering and P. Klint, editors. Algebraic Specification. ACM Press/Addison-Wesley, 1989.

[BJKO99] M.G.J. van den Brand, H.A. de Jong, P. Klint and P.A. Olivier. Efficient Annotated Terms. Submitted 1999.

[BK98] J.A. Bergstra and P. Klint. The Discrete Time ToolBus -a software coordination architecture-. Science of Computer Programming, 31(2-3):205-229, July 1998.

[BKM97] MG.J. van den Brand, T. Kuipers, L. Moonen and P. Olivier. Implementation of a prototype for the new ASF+SDF Meta-Environment. Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications. September 1997.

[BKO99] M.G.J. van den Brand, P. Klint and P.A. Olivier. Compilation and memory management for ASF+SDF. CC '99.

[BKV96] M.G.J. van den Brand, P. Klint and C. Verhoef. Core Technologies for System Renovation. In K.G. Jeffrey, J. Král, and M. Bartosek, editors, SOFSEM '96: Theory and Practice of Informatics, volume 1175 of LNCS, pages 235-255. Springer Verslag, 1996.

[BKV96-b] M.G.J. van den Brand, P. Klint and C. Verhoef. Reengineering needs generic programming language technology. Technical Report P9618, University of Amsterdam, Programming Research Group, 1996.

[BKV98] M.G.J. van den Brand, P. Klint and C. Verhoef. Term Rewriting for Sale. Electronic Notes in Theoretical Computer Science 15 (1998). Available at: http://www.elsevier.nl/locate/entcs/volume15.html

[BSV97] M.G.J. van den Brand, M.P.A. Sellink and C. Verhoef. Obtaining a COBOL grammar from legacy code for reenigineering purposes. In M.P.A. Sellink, editor, Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, electronic Workshops in Computing. Springer verslag, 1997. Available at http://adam.wins.uva.nl/~x/cfn/cfn.html

[BSV98] M.G.J. van den Brand, M.P.A. Sellink and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In Proceedings of the Second Euromicro Conference on Maintenance and Reengineering, pages 11-19, 1998. Available at http://adam.wins.uva.nl/~x/ref.ref.html

[John75] S.C. Johnson. YACC - Yet Another Compiler-Compiler. Technical Report Computer Science No. 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[CHP91] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. Computer Languages, 16(1):97-107, 1991.

[COB] Cobol Language reference on the internet at IBM: http://www.s390.ibm.com/bookmgr-cgi/bookmgr.cmd/BOOKS/IGYLR102/

CONTENTS

[DKV99] A. van Deursen, P. Klint and C.Verhoef. Research issues in the renovation of legacy systems. In J. P. Finance, editor, Fundamental Approaches to Software Engineering, LNCS. Springer-Verslag, 1999. Available at http://adam.wins.uva.nl/~x/etaps/etaps99.html

[Ebb90] W.B.C. Ebbinkhuijsen. COBOL vijfde druk. ISBN 90 14 04560 3.

[Eck98] B Eckel. Thinking in Java. Downloadable from http://www.bruceeckel.com. More technical descriptions can be found at http://java.sun.com

[Em98] Emendo Software Group, the Netherlands. Emendo Y2K White paper, 1998. Available at http://www.emendo.com.

[HKR89] J. Heering, P.R.H. Hendriks, P. Klint and J.Rekers. The syntax definition formalism SDF - Reference manual. SIGPLAN Notices, 24(11):43-75, 1989. Also available at http://www.cwi.nl/~gipe/

[JNI99] Code and documentation regarding the Java Native Interface can be found at http://java.sun.com/products/jdk/1.1/docs/guide/jni/

[JO99] H.A. de Jong and P.A. Olivier. ATerm User Manual. 1999.

[Kli92] P. Klint. A meta-environment for generating programming environments. ACM Transactions on Software Engineering and Methodology 2(2):176-201,1993.

[KN97] K. Nelte. Formulas of First-Order Logic in Distributive Normal Form. Department of Mathematics and Applied Methematics University of Cape Town.

[KR88] B. Kernighan and D.M. Ritchie. The C programming language. ISBN 90-6233-488-1.

[LS86] M.E. Lesk and E. Schmodt. LEX - A lexical analyzer generator. Bell Laboratories, unix programmer's supplementary documents, volume 1 (PS1) edition, 1986.

[RA92] Reasoning Systems, Palo Alto, California. Refine User's Guide, 1992.

[SAV91] W.A Savitch. Pascal third edition. ISBN 0-8053-7450-7.

[SM97] E. Stroo, editor, Microsoft Corporation. Visual Basic 5.0 Programmer's Guide.

[SV98] M.P.A. Sellink and C. Verhoef. An architecture for automated software maintainance. Technical Report P9807, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/~x/asm/asm.html

[Vis97] E. Visser. Phd thesis Syntax Definition for Language Prototyping. ISBN 90-74795-75-7. Programming Research Group, University of Amsterdam.

[Vis97-2] E. Visser. Scanerless Generalized-LR Parsing. Report P9707 August 1997. Programming Research Group, University of Amsterdam.

[Wirth71] N. Wirth. The design of a Pascal compiler, Software-Practice and Experience, 1, 105-133, 1971

Appendices

Appendix A

Interesting parts of the SDF2 definitions of GLF and UIGLF:

module GLF-AriExpressions

```
context-free    syntax
GLF-AddExpr           -> GLF-AriExpr
GLF-MulExpr           -> GLF-AriExpr
GLF-PowerOfExpr       -> GLF-AriExpr
GLF-UnaryExpr         -> GLF-AriExpr
GLF-PrimExpr          -> GLF-AriExpr
GLF-Identifier        -> GLF-AriExpr
GLF-Literal           -> GLF-AriExpr
GLF-FigLit            -> GLF-AriExpr

“add” “(“ GLF-AriExpr “,” GLF-AriExpr “)”> GLF-AddExpr
“sub” “(“ GLF-AriExpr “,” GLF-AriExpr “)”> GLF-AddExpr

“mul” “(“ GLF-AriExpr “,” GLF-AriExpr “)”> GLF-MulExpr
“div” “(“ GLF-AriExpr “,” GLF-AriExpr “)”> GLF-MulExpr

“pow” “(“ GLF-AriExpr “,” GLF-AriExpr “)”> GLF-PowerOfExpr

“neg”   “(“ GLF-AriExpr “)”             -> GLF-UnaryExpr
GLF-PrimExpr                            -> GLF-UnaryExpr

GLF-Identifier      -> GLF-PrimExpr
GLF-Literal         -> GLF-PrimExpr
GLF-FigLit          -> GLF-PrimExpr

UIGLF-AriExpr UIGLF-CHR-PLUS  UIGLF-AriExpr  -> UIGLF-AriExpr
UIGLF-AriExpr UIGLF-CHR-MINUS UIGLF-AriExpr  -> UIGLF-AriExpr
UIGLF-AriExpr UIGLF-CHR-MUL   UIGLF-AriExpr  -> UIGLF-AriExpr
UIGLF-AriExpr UIGLF-CHR-DIV   UIGLF-AriExpr -> UIGLF-AriExpr
UIGLF-AriExpr UIGLF-CHR-POW   UIGLF-AriExpr  -> UIGLF-AriExpr

UIGLF-UnaryExpr                  -> UIGLF-AriExpr

UIGLF-CHR-MINUS UIGLF-UnaryExpr     -> UIGLF-UnaryExpr
UIGLF-PrimExpr        -> UIGLF-UnaryExpr

UIGLF-Identifier    -> UIGLF-PrimExpr
UIGLF-Literal       -> UIGLF-PrimExpr
UIGLF-FigLit        -> UIGLF-PrimExpr
“(“ UIGLF-AriExpr “)”> UIGLF-PrimExpr
```

module GLF-RelExpressions

```
context-free    syntax
GLF-FigLit -> GLF-RelExpr
GLF-AriExpr-> GLF-RelExpr

“lte” “(“ GLF-AriExpr “,” GLF-AriExpr “)” GLF-RelExpr
“gte” “(“ GLF-AriExpr “,” GLF-AriExpr “)” GLF-RelExpr
“lt”   “(“ GLF-AriExpr “,” GLF-AriExpr “)” GLF-RelExpr
“gt”   “(“ GLF-AriExpr “,” GLF-AriExpr “)” GLF-RelExpr
“neq” “(“ GLF-AriExpr “,” GLF-AriExpr “)”> GLF-RelExpr
“eq”   “(“ GLF-AriExpr “,” GLF-AriExpr “)” GLF-RelExpr

“(“ UIGLF-RelExpr “)”> UIGLF-RelExpr
UIGLF-FigLit        -> UIGLF-RelExpr
UIGLF-AriExpr       -> UIGLF-RelExpr
UIGLF-AriExpr UIGLF-RelExprOperator UIGLF-AriExpr    -> UIGLF-RelExpr
```

module GLF-Conditions

```
context-free    syntax

GLF-RelExpr-> GLF-LogExpr
“and” “(“ GLF-LogExpr “,” GLF-LogExpr “)”-> GLF-LogExpr
“or”  “(“ GLF-LogExpr “,” GLF-LogExpr “)”-> GLF-LogExpr
“not” “(“ GLF-LogExpr “)”                -> GLF-LogExpr
GLF-LogExpr           -> GLF-Condition

UIGLF-LogExpr         -> UIGLF-Condition

UIGLF-RelExpr         -> UIGLF-LogExpr

“not” “(“ UIGLF-LogExpr “)”   -> UIGLF-LogExpr

UIGLF-LogExpr “and” UIGLF-LogExpr-> UIGLF-LogExpr
UIGLF-LogExpr “or”  UIGLF-LogExpr-> UIGLF-LogExpr
“(“ UIGLF-LogExpr “)”         -> UIGLF-LogExpr
```

Appendix B

Interesting parts of the GLF Toolbox rewriters:

module RemoveOperators

```
equations

%% Romeove all Ors
%%
[RO1] RemOr( GLF-RelExpr ) = GLF-RelExpr

%% A not around an or: continue
[RO2] RemOr( not(   or(GLF-LogExpr1, GLF-LogExpr2) ) ) =
        not( RemOr(   or(GLF-LogExpr1, GLF-LogExpr2) ) )

%% A not around an and: continue
[RO3] RemOr( not( and(GLF-LogExpr1, GLF-LogExpr2) ) ) =
        not( RemOr( and(GLF-LogExpr1, GLF-LogExpr2) ) )

%% A 'not' around another 'not': remove both
[RO4] RemOr( not( not(GLF-LogExpr) ) ) = RemOr(GLF-LogExpr)

%% We also remove inner Not's
[RO5] RemOr( not( GLF-RelExpr ) ) = RemNot( not(GLF-RelExpr) )

[RO6] RemOr( and( GLF-LogExpr1, GLF-LogExpr2 ) ) =
       and( RemOr( GLF-LogExpr1 ) , RemOr( GLF-LogExpr2 ) )

%% Here he comes: p or q   ->    not( not(p) and not(q) )
%% Notice how the RemOr's are around the newly introduced 'nots' in order to
%% remove possibly added double 'nots'
[RO7] RemOr( or( GLF-LogExpr1, GLF-LogExpr2 ) ) =
        not( and( RemOr(not(GLF-LogExpr1)) , RemOr(not(GLF-LogExpr2)) ) )

%% Remove all nots
%%
[RN1] RemNot( GLF-RelExpr  ) = GLF-RelExpr

[RN2] RemNot( not( and(GLF-LogExpr1, GLF-LogExpr2) ) ) =
        or( RemNot( not(GLF-LogExpr1) ) , RemNot( not(GLF-LogExpr2) ) )

[RN3] RemNot( not(   or(GLF-LogExpr1, GLF-LogExpr2) ) ) =
        and( RemNot( not(GLF-LogExpr1) ), RemNot( not(GLF-LogExpr2) ) )

[RN4] RemNot( not( not( GLF-LogExpr ) ) ) = RemNot( GLF-LogExpr )

[RN5] RemNot( and(GLF-LogExpr1, GLF-LogExpr2) ) =
       and( RemNot(GLF-LogExpr1) , RemNot(GLF-LogExpr2) )

[RN6] RemNot( or(GLF-LogExpr1, GLF-LogExpr2) ) =
       or( RemNot(GLF-LogExpr1) , RemNot(GLF-LogExpr2) )

%% Invert operators
[RN7-1] RemNot( not(   st(GLF-AriExpr1, GLF-AriExpr2) ) ) =
           gte(GLF-AriExpr1, GLF-AriExpr2)
[RN7-2] RemNot( not(   gt(GLF-AriExpr1, GLF-AriExpr2) ) ) =
           lte(GLF-AriExpr1, GLF-AriExpr2)
[RN7-3] RemNot( not( ste(GLF-AriExpr1, GLF-AriExpr2) ) ) =
           gt(GLF-AriExpr1, GLF-AriExpr2)
[RN7-4] RemNot( not( gte(GLF-AriExpr1, GLF-AriExpr2) ) ) =
           lt(GLF-AriExpr1, GLF-AriExpr2)
[RN7-5] RemNot( not(   eq(GLF-AriExpr1, GLF-AriExpr2) ) ) =
           neq(GLF-AriExpr1, GLF-AriExpr2)
[RN7-6] RemNot( not( neq(GLF-AriExpr1, GLF-AriExpr2) ) ) =
           eq(GLF-AriExpr1, GLF-AriExpr2)
```

```
%% Remove all Ands
%%
[RA1] RemAnd( GLF-RelExpr ) = GLF-RelExpr

%% A not around an or: continue
[RA2] RemAnd( not(   or(GLF-LogExpr1, GLF-LogExpr2) ) ) =
        not( RemAnd(   or(GLF-LogExpr1, GLF-LogExpr2) ) )

%% A not around an and: continue
[RA3] RemAnd( not( and(GLF-LogExpr1, GLF-LogExpr2) ) ) =
        not( RemAnd( and(GLF-LogExpr1, GLF-LogExpr2) ) )

%% A 'not' around another 'not': remove both
[RA4] RemAnd( not( not(GLF-LogExpr) ) ) = RemAnd(GLF-LogExpr)

%% We also remove inner not's
[RO5] RemAnd( not( GLF-RelExpr ) ) = RemNot( not(GLF-RelExpr) )

[RA6] RemAnd( or( GLF-LogExpr1, GLF-LogExpr2 ) ) =
       or( RemAnd( GLF-LogExpr1 ) , RemAnd( GLF-LogExpr2 ) )

%% Here he comes: p and q   ->    not( not(p) or not(q) )
%% Notice how the RemAnd's are around the newly introduced 'nots' in order
%% to remove possibly added doubles
[RA7] RemAnd( and( GLF-LogExpr1, GLF-LogExpr2 ) ) =
        not( or( RemAnd(not(GLF-LogExpr1)) , RemAnd(not(GLF-LogExpr2)) ) )
```

module NormalForms

```
equations

[CNF1] move-negs        ( GLF-LogExpr1 ) = GLF-LogExpr2,
          distribute-ors ( GLF-LogExpr2 ) = GLF-LogExpr3
       ====================
         2CNF              ( GLF-LogExpr1 ) = GLF-LogExpr3

[DNF1] move-negs         ( GLF-LogExpr1 ) = GLF-LogExpr2,
          distribute-ands ( GLF-LogExpr2 ) = GLF-LogExpr3
       ====================
         2DNF               ( GLF-LogExpr1 ) = GLF-LogExpr3

%% move-negs move negations inwards (no removal)
%%
[mn1] move-negs(       GLF-RelExpr   ) =        GLF-RelExpr

[mn2] move-negs( not(GLF-RelExpr) ) = not( GLF-RelExpr )

[mn3] move-negs( not( and(GLF-LogExpr1, GLF-LogExpr2) ) ) =
         or(   move-negs( not(GLF-LogExpr1) ) , move-negs( not(GLF-LogExpr2) ) )

[mn4] move-negs( not(or(GLF-LogExpr1, GLF-LogExpr2)) ) =
         and( move-negs( not(GLF-LogExpr1) ), move-negs( not(GLF-LogExpr2) ) )

[mn5] move-negs( not( not( GLF-LogExpr ) ) ) = move-negs( GLF-LogExpr )

[mn6] move-negs( and(GLF-LogExpr1, GLF-LogExpr2) ) =
         and( move-negs(GLF-LogExpr1) , move-negs(GLF-LogExpr2) )

[mn7] move-negs( or(GLF-LogExpr1, GLF-LogExpr2) ) =
         or( move-negs(GLF-LogExpr1) , move-negs(GLF-LogExpr2) )

%% is-and
%%
[ia1]            is-and( and(GLF-LogExpr1, GLF-LogExpr2) ) = yes

[default-ia]  is-and(GLF-LogExpr)  =  no
```

```
%% is-or
%%
[io1]             is-or(  or(GLF-LogExpr1,  GLF-LogExpr2)  )  =  yes

[default-io]   is-or(GLF-LogExpr)   =   no

%%   distribute-ors
%%

%%  or(p,  and(q,r)  )  =  and(  or(p,q)  ,  or(p,r)  )
%%
[do1]  distribute-ors(  or(GLF-LogExpr1,  and(GLF-LogExpr2,  GLF-LogExpr3)  )  )  =
         and(
                    distribute-ors(or(GLF-LogExpr1,  GLF-LogExpr2))  ,
                   distribute-ors(or(GLF-LogExpr1,  GLF-LogExpr3))          )

%%  or(  and(q,r),  p  )  =  and(  or(p,q)  ,  or(p,r)  )
%%
[do2]  distribute-ors(  or(and(GLF-LogExpr2,  GLF-LogExpr3),  GLF-LogExpr1  )  )  =
         and(
                    distribute-ors(or(GLF-LogExpr1,  GLF-LogExpr2))  ,
                   distribute-ors(or(GLF-LogExpr1,  GLF-LogExpr3))          )

%%  All  the  other  cases  can  have  no  'ands'  directly  within  an  or
%%
[default-do4]  distribute-ors(  GLF-LogExpr1  )  =  GLF-LogExpr3,
                     is-and(  GLF-LogExpr3  )  =  NO,
                       distribute-ors(  GLF-LogExpr2  )  =  GLF-LogExpr4,
                     is-and(  GLF-LogExpr4  )  =  NO
                   ====================
                       distribute-ors(  or(  GLF-LogExpr1,  GLF-LogExpr2  )  )  =
                     or(  GLF-LogExpr3,  GLF-LogExpr4  )

[default-do5]  distribute-ors(  GLF-LogExpr1  )  =  GLF-LogExpr3,
                      is-and(  GLF-LogExpr3  )  =  YES,
                        distribute-ors(  GLF-LogExpr2  )  =  GLF-LogExpr4,
                   ====================
                       distribute-ors(  or(  GLF-LogExpr1,  GLF-LogExpr2  )  )  =
                        distribute-ors(  or(  GLF-LogExpr3,  GLF-LogExpr4  )  )

[default-do6]  distribute-ors(  GLF-LogExpr1  )  =  GLF-LogExpr3,
                       distribute-ors(  GLF-LogExpr2  )  =  GLF-LogExpr4,
                     is-and(  GLF-LogExpr4  )  =  YES
                   ====================
                       distribute-ors(  or(  GLF-LogExpr1,  GLF-LogExpr2  )  )  =
                        distribute-ors(  or(  GLF-LogExpr3,  GLF-LogExpr4  )  )

[do8]  distribute-ors(  and(  GLF-LogExpr1,  GLF-LogExpr2  )  )  =
         and(
                    distribute-ors(  GLF-LogExpr1  ),
                   distribute-ors(  GLF-LogExpr2  )      )

%%  since  we  have  already  moved  the  negations  to  the  RelExprs
%%  (with  'move-negs')  they  don't  need  to  be  processed  further
%%  i.e.  a  'not'  is  allways  around  a  'GLF-RelExpr'

[do9]    distribute-ors(  not(  GLF-RelExpr  )  )  =  not(  GLF-RelExpr  )

[do10]  distribute-ors(          GLF-RelExpr     )  =          GLF-RelExpr


%%   distribute-ands
%%

%%  and(p,  or(q,r)  )  =  or(  and(p,q)  ,  and(p,r)  )
%%
[da1]  distribute-ands(  and(GLF-LogExpr1,  or(GLF-LogExpr2,GLF-LogExpr3)  )  )  =
```

```
         or(
                  distribute-ands(and(GLF-LogExpr1,  GLF-LogExpr2))  ,
                 distribute-ands(and(GLF-LogExpr1,  GLF-LogExpr3))       )

%%  and(  or(q,r),  p  )  =  or(  and(p,q)  ,  and(p,r)  )
%%
[da2]  distribute-ands(  and(or(GLF-LogExpr2,  GLF-LogExpr3),GLF-LogExpr1  )  )  =
         or(
                  distribute-ands(and(GLF-LogExpr1,  GLF-LogExpr2))  ,
                 distribute-ands(and(GLF-LogExpr1,  GLF-LogExpr3))       )

%%  All  the  other  cases  can  have  no  'ors'  within  an  'and'
%%
[default-da4]  distribute-ands(    GLF-LogExpr1  )  =  GLF-LogExpr3,
                    is-or( GLF-LogExpr3 ) = NO,
                     distribute-ands(   GLF-LogExpr2 ) = GLF-LogExpr4,
                   is-or( GLF-LogExpr4 ) = NO
                  ====================
                      distribute-ands(   and( GLF-LogExpr1,  GLF-LogExpr2 ) ) =
                    and( GLF-LogExpr3,  GLF-LogExpr4 )

[default-da5]  distribute-ands(    GLF-LogExpr1  )  =  GLF-LogExpr3,
                     is-or( GLF-LogExpr3 ) = YES,
                      distribute-ands(   GLF-LogExpr2 ) = GLF-LogExpr4,
                  ====================
                      distribute-ands(  and( GLF-LogExpr1,  GLF-LogExpr2 ) ) =
                       distribute-ands(  and(  GLF-LogExpr3,  GLF-LogExpr4 ) )

[default-da6]  distribute-ands(    GLF-LogExpr1  )  =  GLF-LogExpr3,
                      distribute-ands(   GLF-LogExpr2 ) = GLF-LogExpr4,
                    is-or( GLF-LogExpr4 ) = YES
                  ====================
                      distribute-ands(  and( GLF-LogExpr1,  GLF-LogExpr2 ) ) =
                       distribute-ands(  and(  GLF-LogExpr3,  GLF-LogExpr4 ) )

[da8]  distribute-ands(  or(  GLF-LogExpr1,  GLF-LogExpr2  )  )  =
         or(
                  distribute-ands(  GLF-LogExpr1 ),
                 distribute-ands(  GLF-LogExpr2 )       )

%%  since  we  have  already  moved  the  negations  to  the  RelExprs
%%  (with  'move-negs')  they  don't  need  to  be  processed  further
%%  i.e.  a  'not'  is  allways  around  a  'GLF-RelExpr'

[da9]    distribute-ands(  not(  GLF-RelExpr  )  )  =  not(  GLF-RelExpr  )

[da10]  distribute-ands(          GLF-RelExpr     )  =          GLF-RelExpr
```

Appendix C

Interesting parts of the GLF to UIGLF translator:

module GLF-AriExpressions

```
equations

%% UIGLF-AriExpr
%%
[UIGLF-AE1]  UIGLF-AriExpr  (  add(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b-add(  GLF-AriExpr1  )  +
                    UIGLF-AriExpr-b-add(  GLF-AriExpr2  )

[UIGLF-AE2]  UIGLF-AriExpr  (  sub(GLF-AriExpr1,GLF-AriExpr2)  )  =
                     UIGLF-AriExpr-b-sub(  GLF-AriExpr1  )  -
                     UIGLF-AriExpr-b-sub(  GLF-AriExpr2  )

[UIGLF-AE3]  UIGLF-AriExpr  (  mul(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b-mul(  GLF-AriExpr1  )  *
                    UIGLF-AriExpr-b-mul(  GLF-AriExpr2  )

[UIGLF-AE4]  UIGLF-AriExpr  (  div(GLF-AriExpr1,GLF-AriExpr2)  )  =
                     UIGLF-AriExpr-b-div(  GLF-AriExpr1  )  /
                     UIGLF-AriExpr-b-div(  GLF-AriExpr2  )

[UIGLF-AE5]  UIGLF-AriExpr  (  pow(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b(  GLF-AriExpr1  )  ^
                    UIGLF-AriExpr-b(  GLF-AriExpr2  )

[UIGLF-AE6] UIGLF-UnaryExpr( GLF-AriExpr ) = UIGLF-UnaryExpr
               ====================
                UIGLF-AriExpr  (  GLF-AriExpr  )  =  UIGLF-UnaryExpr

%% UIGLF-AriExpr-b
%%
%%  no  brackets  around  UnaryExpr
[UIGLF-AEb1] UIGLF-UnaryExpr( GLF-AriExpr ) = UIGLF-UnaryExpr
                ====================
                 UIGLF-AriExpr-b  (  GLF-AriExpr  )  =  UIGLF-UnaryExpr

[default-UIGLF-AEb]  UIGLF-AriExpr-b(  GLF-AriExpr  )  =
                           (  UIGLF-AriExpr(  GLF-AriExpr  )  )


%%  UIGLF-AriExpr-b-add           Brackets,  except  for  addexpr
%%
[UIGLF-AEb-add1]  UIGLF-AriExpr-b-add  (  add(GLF-AriExpr1,GLF-AriExpr2)  )  =
                           UIGLF-AriExpr(  add(GLF-AriExpr1,GLF-AriExpr2)  )

[UIGLF-AEb-add1]  UIGLF-AriExpr-b-add  (  sub(GLF-AriExpr1,GLF-AriExpr2)  )  =
                           UIGLF-AriExpr(  sub(GLF-AriExpr1,GLF-AriExpr2)  )

[default-UIGLF-AEb-add]  UIGLF-AriExpr-b-add(  GLF-AriExpr  )  =
                                 UIGLF-AriExpr-b(  GLF-AriExpr  )


%%  UIGLF-AriExpr-b-mul           Brackets,  except  for  mulexpr
%%
[UIGLF-AEb-add1]  UIGLF-AriExpr-b-mul  (  mul(GLF-AriExpr1,GLF-AriExpr2)  )  =
                           UIGLF-AriExpr(  mul(GLF-AriExpr1,GLF-AriExpr2)  )

[UIGLF-AEb-add1]  UIGLF-AriExpr-b-mul  (  div(GLF-AriExpr1,GLF-AriExpr2)  )  =
                           UIGLF-AriExpr(  div(GLF-AriExpr1,GLF-AriExpr2)  )
```

```
[default-UIGLF-AEb-add]  UIGLF-AriExpr-b-mul(  GLF-AriExpr  )  =
                                  UIGLF-AriExpr-b(  GLF-AriExpr  )

%% PrimExpr
%%
[UIGLF-PE1]  UIGLF-PrimExpr  (  GLF-Identifier  )  =
                    UIGLF-Identifier  (  GLF-Identifier  )

[UIGLF-PE2]  UIGLF-PrimExpr  (  GLF-Literal  )  =  UIGLF-Literal(  GLF-Literal  )

[UIGLF-PE3]  UIGLF-PrimExpr  (  GLF-FigLit  )  =  UIGLF-FigLit     (  GLF-FigLit  )

%% UnaryExpr
%%
[UIGLF-UE1]  UIGLF-UnaryExpr  (  GLF-AriExpr  )  =
                  UIGLF-PrimExpr   (  GLF-AriExpr  )

[UIGLF-UE3]  UIGLF-UnaryExpr  (  neg(GLF-AriExpr)  )  =
                  -  UIGLF-UnaryExpr  (  GLF-AriExpr  )
```

module GLF-RelExpressions

```
equations

%%  By  Alex  van  den  Bergh  july  1999

%%  RelExpr
%%
[UIGLF-RE1]  UIGLF-RelExpr  (  GLF-AriExpr  )  =  UIGLF-AriExpr-b  (  GLF-AriExpr  )

[UIGLF-RE2]  UIGLF-RelExpr  (  lte(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b(  GLF-AriExpr1  )  <=
                    UIGLF-AriExpr-b(  GLF-AriExpr2  )

[UIGLF-RE3]  UIGLF-RelExpr  (  gte(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b(  GLF-AriExpr1  )  >=
                    UIGLF-AriExpr-b(  GLF-AriExpr2  )

[UIGLF-RE4]  UIGLF-RelExpr  (  lt(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b(  GLF-AriExpr1  )  <
                    UIGLF-AriExpr-b(  GLF-AriExpr2  )

[UIGLF-RE5]  UIGLF-RelExpr  (  gt(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b(  GLF-AriExpr1  )  >
                    UIGLF-AriExpr-b(  GLF-AriExpr2  )

[UIGLF-RE6]  UIGLF-RelExpr  (  neq(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b(  GLF-AriExpr1  )  <>
                    UIGLF-AriExpr-b(  GLF-AriExpr2  )

[UIGLF-RE7]  UIGLF-RelExpr  (  eq(GLF-AriExpr1,GLF-AriExpr2)  )  =
                    UIGLF-AriExpr-b(  GLF-AriExpr1  )  ==
                    UIGLF-AriExpr-b(  GLF-AriExpr2  )

%%  RelExpr-b  Brackets  only  when  relExpr  (with  relational  operator)
%%  so  no  brackets  around  identifiers,  literals  etc.

[UIGLF-RE-b1]  UIGLF-RelExpr-b  (  GLF-AriExpr  )  =
                    UIGLF-AriExpr-b  (  GLF-AriExpr  )

[UIGLF-RE-default]   UIGLF-RelExpr-b  (  GLF-RelExpr  )  =
                            (  UIGLF-RelExpr(  GLF-RelExpr  )  )
```

module GLF-Conditions

```
equations

%% LogExpr
%
[UIGLF-LE1] UIGLF-LogExpr-b-or ( GLF-LogExpr1 ) = UIGLF-LogExpr1,
              UIGLF-LogExpr-b-or ( GLF-LogExpr2 ) = UIGLF-LogExpr2
             ====================
              UIGLF-LogExpr ( or(GLF-LogExpr1,GLF-LogExpr2) )
                                =  UIGLF-LogExpr1 or UIGLF-LogExpr2

[UIGLF-LE2] UIGLF-LogExpr-b-and ( GLF-LogExpr1 ) = UIGLF-LogExpr1,
              UIGLF-LogExpr-b-and ( GLF-LogExpr2 ) = UIGLF-LogExpr2
             ====================
              UIGLF-LogExpr ( and(GLF-LogExpr1,GLF-LogExpr2) )
                                =  UIGLF-LogExpr1 and UIGLF-LogExpr2

[UIGLF-LE2] UIGLF-LogExpr ( GLF-LogExpr ) = UIGLF-LogExpr
             ====================
              UIGLF-LogExpr ( not(GLF-LogExpr) ) =  not( UIGLF-LogExpr )

[UIGLF-LE2] UIGLF-RelExpr ( GLF-RelExpr ) = UIGLF-LogExpr
             ====================
              UIGLF-LogExpr ( GLF-RelExpr ) = UIGLF-LogExpr


%% LogExpr-b = Expr with Brackets
%%  except  for  not  because  that  already  has  brackets

[UIGLF-LEb1] UIGLF-LogExpr-b( not( GLF-LogExpr ) ) =
               UIGLF-LogExpr( not( GLF-LogExpr ) )

[default-UIGLF-LEb] UIGLF-LogExpr-b( GLF-LogExpr ) =
                          ( UIGLF-LogExpr( GLF-LogExpr ) )


%% LogExpr-b-or = Expr with Brackets except for or
%
[UIGLF-LEb-or1] UIGLF-LogExpr(or(GLF-LogExpr1,GLF-LogExpr2)) = UIGLF-LogExpr
                  ====================
                    UIGLF-LogExpr-b-or ( or(GLF-LogExpr1,GLF-LogExpr2) ) =
                   UIGLF-LogExpr

[default-UIGLF-LEb-or] UIGLF-LogExpr-b     ( GLF-LogExpr ) = UIGLF-LogExpr
                              ====================
                               UIGLF-LogExpr-b-or ( GLF-LogExpr ) = UIGLF-LogExpr


%% LogExpr-b-and = Expr with Brackets except for and
%
[UIGLF-LEb-and1] UIGLF-LogExpr(and(GLF-LogExpr1,GLF-LogExpr2))=UIGLF-LogExpr
                   ====================
                     UIGLF-LogExpr-b-and ( and(GLF-LogExpr1,GLF-LogExpr2) ) =
                    UIGLF-LogExpr

[default-UIGLF-LEb-and] UIGLF-LogExpr-b       ( GLF-LogExpr ) = UIGLF-LogExpr
                                ====================
                                 UIGLF-LogExpr-b-and ( GLF-LogExpr ) = UIGLF-LogExpr

%% Condition
%
[UIGLF-LOE1] UIGLF-LogExpr    ( GLF-LogExpr ) = UIGLF-LogExpr
              ====================
               GLF2UIGLF-Condition ( GLF-LogExpr ) = UIGLF-LogExpr
```

Appendix D

The Java part of the Meta-adapter, to illustrate how the TRS is connected to other systems and to show how it can be used.

File TRS.java

```
package     COM.triloc.javatrs;

/**  Class  TRS.
  */
public  class  TRS
{
   public  static  final  String  JAVATRS_DLL   “c:\\javatrs”;
   public  static  final  String  COBOL_PARSE_TABLE “c:\\batch.tbl”;
   public  static  final  String  TEST_FUNCTION “MCB2UIGLF-Condition”;
   public  static  final  int        TEST_ID  =  0;
   public  static  final  String  TEST_CONDITION “RECORD::FIELD = ‘STRING’”;

   static
   {
      System.loadLibrary(JAVATRS_DLL);
   }

   /**  Native  methods
     *    actually  implemented  in  C
     */
   private   static   native   String _swingTerm(String   p_istr,   String   p_ptable);
   private   static   native   void _loadParseTable(String      p_ptable);
   private   static   native   int _errorTest();
   private   static   native   String _getErrorMsg();

   /**   Default   constructor.
     */
   public   TRS()
   {
   }

   /**  Load  a  Parse  Table
     */
   public   void   loadParseTable(String   p_ptable)
      throws   TRSException,   FileNotFoundException,   IOException
   {
      File   l_file   =   new   File(p_ptable);
      FileReader   l_filereader   =   new   FileReader(l_file);
      l_filereader.close();

      _loadParseTable(p_ptable);

      /*  Check  whether  an  error  has  occured  (in  native  code)
        *  while  loading  the  parse  table
        */
      if(_errorTest()   !=   0)
      {
         /*  If  an  error  has  occured  get  the  message
           *  and  throw  it  in  an  exception
           */
         throw  new  TRSException(  _getErrorMsg()  );
      }
   }
```

```
   /**  Rewrite  method  with  paramters.
     */
   public  String  rewriteTerm(String  p_func,  int  p_id,  String  p_cond,
                    String  p_ptable,TRSOption  p_option)
      throws  TRSException
   {
      //  get  input  string
      //  write  it  in  the  right  format  for  the  rewriter
      String  l_istr  =  buildInputString(p_func,  p_id,  p_cond,  p_option);

      //  get  output  string
      String  l_ostr  =  _rewriteTerm(l_istr,  p_ptable);

      //  compress  output  string
      //  (remove  redundant  spaces,  returns  etc.  introduced  by  the  TRS
      l_ostr  =  _compressString(l_ostr);

      if(_errorTest()  ==  0)
      {
         //  no  error  has  occured
         return  l_ostr;
      }

      throw  new  TRSException(  _getErrorMsg()  );
   }


   /**  Method  that  builds  the  input  string  for  the  TRS  from  the
     *  function  name  the  id  and  the  condition.
     */
   public  static  String  buildInputString(String  p_func,  int  p_id,
                           String  p_cond,TRSOption  p_option)
   {
      return  p_func  +  "  (  "  +  p_id  +  "  {  "  +  p_cond  +  "  }  ,  "  +
               p_option.getTextRepr()  +  "  )  ";
   }

   /**  Method  that  compresses  the  string  parameter.
     */
   private  String  _compressString(String  p_str)
   {
      StringBuffer  l_result  =  new  StringBuffer();

      for(int  l_i  =  0,  l_len  =  p_str.length();  l_i  <  l_len;  l_i++)
      {
         char  l_c  =  p_str.charAt(l_i);

         boolean  l_space  =  false;

         while(l_i  <  l_len  &&
               (l_c  ==  '  '  ||  l_c  ==  '\r'  ||  l_c  ==  '\n'  ||  l_c  ==  '\t'))
         {
            l_i++;
            l_c  =  p_str.charAt(l_i);
            l_space  =  true;
         }

         if(l_space  ==  true)  l_result.append('  ');

         l_result.append(l_c);
      }

      return        l_result.toString().trim();
   }

}
```