

TECHNISCHE UNIVERSITEIT EINDHOVEN  
Department of Mathematics and Computer Science

## Defining metrics for STL-code

By  
Bart Seghers

Supervisors:

**Prof. dr. M.G.J. van den Brand**  
Department of Mathematics and Computing Science  
Software Engineering and Technology Group  
Eindhoven, University of Technology, NL

**ir. F. van den Berk**  
Technology Manager Controls  
Vanderlande Industries Nederland B.V.

13 March 2006 - 18 December 2006

## Abstract

This thesis defines metrics that signal the need for preventive refactoring of statement list components (STL components). These metrics can be used to keep an STL component reusable and maintainable by warning when the internal structure of that component becomes to complex.

It appears that no research had been conducted on STL metrics before and that no tools were available for the collection of metrics either. Therefore, metrics used on other programming languages were analyzed and used to define a suitable metrics set for STL.

In order to test the useability of certain metrics, a tool called AWL Analyzer was developed using rapid prototyping. This tool extracts, stores, and orders numerous metrics from STL components and visualizes the communication structures used between those components. AWL Analyzer was constructed iteratively based on feedback of the PLC programmers on prototypes of the tool.

Via a questionnaire, the perceived complexities of a number of STL components were retrieved from the PLC programmers. These were used to compare to a number of metrics extracted by the AWL Analyzer. Based on this comparison, we concluded that there are two types of complexity for an STL component: internal complexity and interaction complexity. The best way to measure internal complexity is with the McCabe complexity measure; Whereas the best way to measure interaction complexity is with information flow metrics. Both metrics can help in defining STL components with a high potential for refactoring.

Finally, complexity trends in STL code were addressed. These show that some components have steady complexity levels while others have increasing complexity levels over time. This makes identifying error-prone components (with the highest priorities for refactoring) more accurate than only measuring current complexity levels.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Background . . . . .	4
1.2	Research question and objectives . . . . .	4
1.3	Deliverables . . . . .	5
1.4	Outline . . . . .	5
1.5	Acknowledgements . . . . .	5
<b>2</b>	<b>Literature research</b>	<b>6</b>
2.1	Software maintenance . . . . .	6
2.2	Research conducted by others on metrics for STL . . . . .	7
2.3	Code metrics available in literature . . . . .	7
2.4	Conclusions . . . . .	9
<b>3</b>	<b>STL usage at Vanderlande</b>	<b>10</b>
3.1	Definitions PLC and STL . . . . .	10
3.2	Standard software architecture . . . . .	10
3.3	Enabling the reuse of components by using blocks . . . . .	14
3.4	Structure components . . . . .	15
3.5	Structure STL instructions . . . . .	17
3.6	Metrics to use for STL . . . . .	20
3.7	Availability of tools who can be used for the collection of metrics . . . . .	20
<b>4</b>	<b>AWL Analyzer</b>	<b>22</b>
4.1	Scanning and parsing STL . . . . .	22
4.2	Collecting metrics from .awl-files . . . . .	23
4.2.1	Non-interaction metrics and revision detection . . . . .	24
4.2.2	Interaction detection . . . . .	25
4.2.3	Sorting tables . . . . .	27
4.3	Representing communication structure in graphs . . . . .	28
4.3.1	Standard graph drawing . . . . .	28
4.3.2	Performance problems for graph drawing . . . . .	30
4.3.3	Coloring dependencies and drawing subgraphs . . . . .	31
4.3.4	Draw all interactions with node . . . . .	32
<b>5</b>	<b>Questionnaire</b>	<b>33</b>
5.1	Questions . . . . .	33
5.2	Results questionnaire . . . . .	35
5.3	Correlation results AWL-Analyzer and questionnaire . . . . .	38
5.3.1	Hypothesis test simple linear regression . . . . .	38
5.3.2	Correlation . . . . .	39
5.3.3	Conclusions correlation results . . . . .	41
<b>6</b>	<b>Trends complexity in code</b>	<b>43</b>
6.1	McCabe and interaction trend graphs 19 components . . . . .	43
6.2	Use AWL Analyzer to visualize structure changes in component . . . . .	47
6.3	Conclusions . . . . .	48
<b>7</b>	<b>Conclusions</b>	<b>49</b>

<b>8 Recommendations</b>	<b>51</b>
<b>9 Evaluation</b>	<b>53</b>
9.1 Change in assignment . . . . .	53
9.2 Rapid prototyping process . . . . .	53
9.3 Usability results . . . . .	53
<b>10 Bibliography</b>	<b>54</b>
<b>A Regression analysis on perceived complexity</b>	<b>56</b>
A.1 Perceived complexity - McCabe . . . . .	56
A.2 Perceived complexity - Lines Of Code . . . . .	57
A.3 Perceived complexity - Revisions . . . . .	58
A.4 Perceived complexity - Comment ratio . . . . .	59
A.5 Perceived complexity - MerkerIO . . . . .	60
A.6 Perceived complexity - FanIO . . . . .	61
A.7 Perceived complexity - Instantiations . . . . .	62
A.8 Perceived complexity - Interaction . . . . .	63

# 1 Introduction

## 1.1 Background

The background for this assignment was presented on the website of *Vanderlande* <http://www.vanderlande.nl>:

*The software layer that directly controls our systems in real time is often implemented on Programmable Logic Controllers (PLCs). The complexity and functionality of this software is growing, so we decided to develop and maintain a re-usable software platform for our PLC-based equipment control. Many projects and engineers contribute to this software platform, while a core team of specialists, headed by a lead-architect, guards its quality and architecture.*

*One of the risks of any software platform with a long intended lifetime is erosion of its internal structure. This causes maintainability and reliability problems, leading to more effort and longer lead-times to realize our systems.*

*We want to fight software erosion by regular preventive refactoring. To support this process, we need metrics that signal the need for preventive refactoring. We want to continuously measure the quality of our software platform with these metrics, and define 'thresholds' for refactoring based on experiences with these metrics.*

*Another possibility is to use these metrics to verify the quality of new contributions or changes to the software platform.*

## 1.2 Research question and objectives

Vanderlande wants to fight software erosion of the internal structure of the PLC software platform by regular preventive refactoring.

The main objective for this assignment therefore is to define metrics that signal the need for preventive refactoring. This objective can be split up in three major sub-tasks and -objectives:

- **To study earlier research in the field of maintainability of PLC code and related metrics. And, if research was conducted, can this be used to improve the maintainability of PLC code at Vanderlande?**
- **To study the architecture of the PLC software platform, learn what determines its maintainability, and define metrics based on these insights.**
- **To specify tools to collect these metrics automatically (these are to be implemented in later student assignments).**

The third task or objective was addressed more thoroughly in this assignment. A tool to collect the metrics was written from scratch.

### 1.3 Deliverables

Initially, the end product in this assignment would only be a report analyzing and answering the questions and objectives from section 1.2. The tool written during this assignment became very useful however. Therefore, the final version of the tool including its source code and documentation also became deliverables.

In detail, the deliverables are:

- Report analyzing and answering questions and objectives from section 1.2.
- AWL Analyzer v1.0 (tool to collect metrics from STL code)
- Source code AWL Analyzer v1.0
- Architectural design AWL Analyzer v1.0
- Detailed design AWL Analyzer v1.0
- Software user manual AWL Analyzer v1.0

### 1.4 Outline

Chapter 2 addresses the literature research conducted in this assignment. It includes the question whether research has been conducted on maintainability of PLC code as well as metrics used on other (non-PLC) code. Chapter 3 explains the language STL (PLC-code) such as used by Vanderlande and defines usable metrics for the STL language in order to determine maintainability. Chapter 4 discusses the AWL Analyzer, a tool developed in this assignment to collect metrics for STL. Chapter 5 contains the questionnaire used to verify the results (metrics) produced by the AWL Analyzer with the perceptions of the PLC-programmers. Chapter 6 explains the process of expanding STL maintainability analysis by analyzing complexity trends of code metrics. Finally, chapters 7, 8, and 9 contain the conclusions, recommendations, and evaluation for this assignment.

### 1.5 Acknowledgements

First of all, I would like to thank my supervisors. Mark van den Brand from TU/e, and Frank van den Berk from Vanderlande Industries Nederland B.V.. They provided a lot of feedback on the sections I wrote, and they were able to guide me in the right direction when necessary. Secondly, I would like to thank Tim van Hassel and Gert Maas from Vanderlande Industries. They taught me a lot about PLCs, STL, and the way they are used at Vanderlande Industries. Furthermore, they provided useful feedback on the prototypes of AWL Analyzer and on the ideas for collecting certain metrics. All four were usually able to fit a couple of minutes in their busy schedules right away to answer my questions. For me this was a very pleasant way of communication which led to a high productivity. Finally, I would like to thank all PLC programmers who took the time to fill in a questionnaire for this assignment.

## 2 Literature research

This section gives an overview of the relevant literature on software maintenance and on code metrics. The definition of software maintenance will be addressed first, including different categories of maintenance and the usage of these categories at Vanderlande. Afterwards, the question what kind of research on STL metrics has already been done will be addressed. Finally, general code metrics will be presented.

### 2.1 Software maintenance

IEEE defines software maintenance as:

“The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.” [IEEE93]

This definition explicitly incorporates three major categories of software maintenance: corrective maintenance, adaptive maintenance, and perfective maintenance. Another major category (proposed by IEEE as well [IEEE93]) is preventive maintenance. A short description/explanation for each of those four major categories will be presented now.

- Corrective maintenance: fixing actual errors in a software product.
- Adaptive maintenance: alterations made to the software product as a result of changes in the environment of the software product.
- Perfective maintenance: alterations made to a software product to meet the evolving and/or expanding needs of the user.
- Preventive maintenance: alterations made to a software product to make it easier to conduct other types of maintenance (corrective, adaptive, and perfective) in the future.

For STL components at Vanderlande, the first three categories (corrective, adaptive, and perfective) of maintenance are used. The dominant maintenance category differs from component to component. For components that are frequently reused, the dominant category will usually be adaptive maintenance or perfective maintenance. New and/or complex components usually need more corrective maintenance.

At the moment of writing, no preventive maintenance takes place for the STL platform at Vanderlande. This is the reason for this assignment, to help incorporate this type of maintenance in the maintenance process at Vanderlande.

## 2.2 Research conducted by others on metrics for STL

Based on thorough searches in

- IEEE-IEE Electronic Library (IEL) (<http://ieeexplore.ieee.org>)
- CiteSeer.IST Scientific Literature Digital Library (<http://citeseer.ist.psu.edu>)
- PLC community fora like PLCS.net (<http://www.plctalk.net/qanda>)
- Siemens product support (<http://support.automation.siemens.com>)
- Google (<http://www.google.com>)

no indication was found that anything of value has been done in the field of complexity-metrics on PLC-code (STL in particular) yet.

One of the main reasons probably is that the STL community is relatively small compared to communities of programming languages like C, C++, C#, Java, Fortran, Pascal, etcetera.

The metrics set therefore had to be built from scratch in this assignment. Some existing complexity metrics used in other programming languages could nevertheless be used as starting points. Furthermore, input and validation of the proposed metrics by PLC programmers (at Vanderlande) were very useful in defining ‘points of interest’ in STL-code.

## 2.3 Code metrics available in literature

The basis for software measures and software measurement was established in the sixties and mainly in the seventies [RH68]. Based on this work, more and more sophisticated measures emerged in the following decennia [Zuse06].

Today, a wide range of software metrics can be found in literature. Some metrics are more sophisticated than others. Some can be used for basically any programming language/project while others are more language-specific.

Software metrics can be divided in two main groups: process metrics and code metrics [Pfl93]. Process metrics include metrics on for example costs and schedules for a project. These metrics are very important, but go beyond the scope of this assignment. Code metrics include metrics like: file size/lines of code, comment ratio, revision metrics, cyclomatic complexity, information flow metrics (Henry & Kafura), Halstead complexity, Bowles metrics, Ligier metrics, Troy & Zweben metrics, and ABC metrics [RD05], [Kit88], [Jon94], [Dav05]. Code metrics are the type of metrics Vanderlande was looking for. The metrics used in this assignment (file size/lines of code, comment ratio, revision metrics, cyclomatic complexity, and information flow metrics) will be addressed in more detail now.

### File size/LOC

A common basis for estimation in a software project (effort to construct/maintain) is the LOC or lines of code metric [RSM06]. More and more complex functionality usually requires more lines of code. It is relatively easy to count the lines

of code for a file or project, the process is easy to automate and the lines of code metric is intuitive [WikL06]. There are some disadvantages to the lines of code metric. There is a lack of cohesion with the functionality of a piece of code, the number of lines of code for a given functionality can differ from programmer to programmer and there are numerous ways to count<sup>1</sup>. A metric like cyclomatic complexity (addressed later) has less of these disadvantages.

#### **Amount of comment**

A NASA software study suggests that modules with higher comment ratios are, on average, more reusable than others [MAL05]. It is easier to understand what a piece of code is doing when it is well documented. As with the lines of code metric, there are numerous ways to define comment ratio. Some examples are: total lines of code/total lines of comments, the characters of code/characters of comments and lines containing code/lines containing comments. The one is more detailed, but usually slightly more complex to collect than the other.

#### **Metrics on Revisions**

A number of metrics can be collected on revisions. Some examples are amount of revisions, frequency (and distribution) of revisions, total amount of programmers for a module and size of the revisions. A high amount and high frequency of revisions can indicate an error-prone module. A high amount of programmers (a lot of different programming styles) can make the module more complex and harder to understand.

#### **Cyclomatic (McCabe) complexity**

One of the most widely used and accepted software metrics is Cyclomatic Complexity (CC) [RD05]. This metric was introduced by McCabe in 1976 and is therefore often called McCabe complexity. The metric ‘measures’ the amount of linearly independent paths through a piece of code (function, module, etcetera). It represents the number of test cases needed to unit test a piece of code [CMU06].

Cyclomatic complexity is defined by [McC76]:

$$CC = e - n + 2p$$

where E is the number of edges, N the number of nodes and P the number of connected components in a flow graph. It is further demonstrated that the amount of linearly independent paths through a piece of code (CC) is the amount of decision points in that piece of code + 1. So,  $CC = NDP + 1$ , where NDP is the number of decision points [Wey88].

The higher the cyclomatic complexity, the more likely that keeping track of the behavior of the piece of code is difficult.

#### **Information flow (FanIO) metrics**

Information flow metrics, developed by Henry and Kafura, measure the links among functions/procedures/components by ‘measuring’ the information flow

---

<sup>1</sup>including/excluding comment lines/empty lines, count one statement over multiple lines as one line, etcetera

between them [HK84]. Henry and Kafura defined that errors in procedures with a high information flow are more likely to occur. Furthermore, these procedures usually are more complex and more difficult to maintain. A high information flow can therefore indicate a change-prone and error-prone procedure [Kit88].

The IEEE 982.2 formula for informational complexity (IFC) is [IEEE88]:

$$\text{IFC} = (\text{fanin} * \text{fanout})^2 \text{ with:}$$

- fanin = local flows into a procedure + number of data structures from which the procedure retrieves data.
- fanout = local flows from a procedure + number of data structures that the procedure updates.

IFC basically represents the number of paths that can go through the procedure/function/component in question (or at least an upper bound for that).

## 2.4 Conclusions

McCabe complexity and information flow metrics are usually better indicators for complexity than Halstead complexity and Lines Of Code [Jon94]. It would therefore be useful if both McCabe and information flow metrics would be collected for STL components.

### 3 STL usage at Vanderlande

This section starts with the definitions of Programmable Logic Controllers (PLCs) and Statement Lists (STL) as well as how PLCs and STL are used at Vanderlande Industries. Furthermore, it addresses the software architecture used for the STL code at Vanderlande and the implementation of this architecture.

#### 3.1 Definitions PLC and STL

A Programmable Logic Controller (PLC) is a small computer used for the automation of processes like control of machinery or factory production lines [WikP06]. The PLC usually consists of a microprocessor, static memory to store the program to be executed on the PLC (for example written in statement list), work memory used during execution of the program and a number of inputs and outputs. The inputs and outputs are used to connect a number of sensors and actuators to the PLC in order to ‘read’ and ‘control’ the process. PLC’s are usually packaged and designed for extensive temperature ranges, dirty or dusty conditions, immunity to electrical noise, mechanically more rugged and resistant to vibration and impact and their Mean Time To Repair (MTTR) is usually extremely low. This makes them more suitable for controlling a certain process than a ‘regular’ computer [WikP06].

At Vanderlande, SIMATIC S7-300 and SIMATIC S7-400 PLCs are used, depending on the size of a project. The programming language used for these PLCs is “Statement List”.

“Statement List” (STL) (or “anweisungs liste” (AWL) in German) is a textual programming language developed by Siemens. STL corresponds to the Instruction List (IL) language defined in the International Electrotechnical Commission’s standard IEC 61131-3. Instruction list uses very simple instructions similar to the instructions of mnemonics programming languages. The syntax for statements in STL is similar to the syntax of assembler languages. It consists of instructions followed by addresses on which the instructions act. There are more than 130 basic instructions and a wide range of addresses available [Sie98]. For a more detailed overview of instructions and addresses in STL, please see the SIMATIC S7 Reference manual [Sie98].

At Vanderlande, the programming software package STEP 7 (S7) is used for programming in STL. This package provides editing, compiling, testing/debugging, and repository functionality for STL. The package saves all components coded in STL as plain text in files with extensions ‘.awl’. Furthermore, S7 adds information like component type, component name, amount of revisions, revision history, author and component family as comment in the header of these files. This is useful for extracting information from these files later on.

#### 3.2 Standard software architecture

In July 2003, Vanderlande Industries started the Standard Software Structure (SSS) project, defined and guarded by the *S7 team*. This project aims at build-

ing software using a standard architecture (decomposition, functions, and interfaces), standard rules, concepts, tools, and documentation according to the SSS.

Vanderlande Industries achieves this by building their PLC software as reusable components [SST05]. These components are stored in separate files, containing strongly related functionality and communicate via well-defined interfaces. The structure of an application is built according the architecture presented in Figure 3.1. The meaning of this figure will be described in the rest of this section.

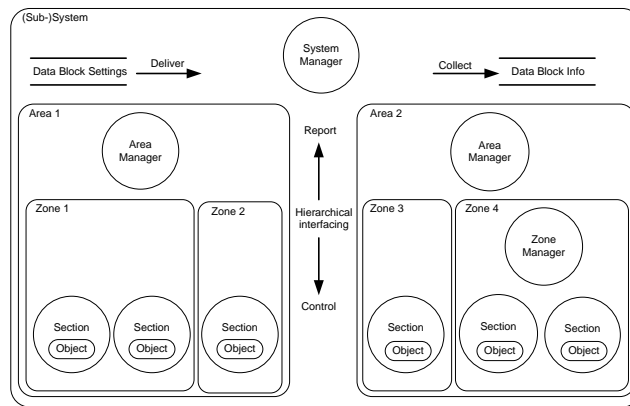


Figure 3.1: Software structure overview

As can be seen, several software layers are present in this architecture:

- System layer
- Area layer
- Zone layer
- Section layer
- Object layer

Each of these layers consist of one or more (reusable) components and one or more manager components. A manager component controls one or more components in the layer below and therefore combines these components into a certain functionality.

In order to clarify the scope of each layer, the definitions of the layers will be explained first. This is done based on an example. Afterwards, the interfacing between the layers and the interfacing between the components in the same layer will be addressed.

### Layer definitions

The scope of each layer will be addressed based on an example of a luggage

check-in system at an airport. The layout of such a system is presented in Figure 3.2.

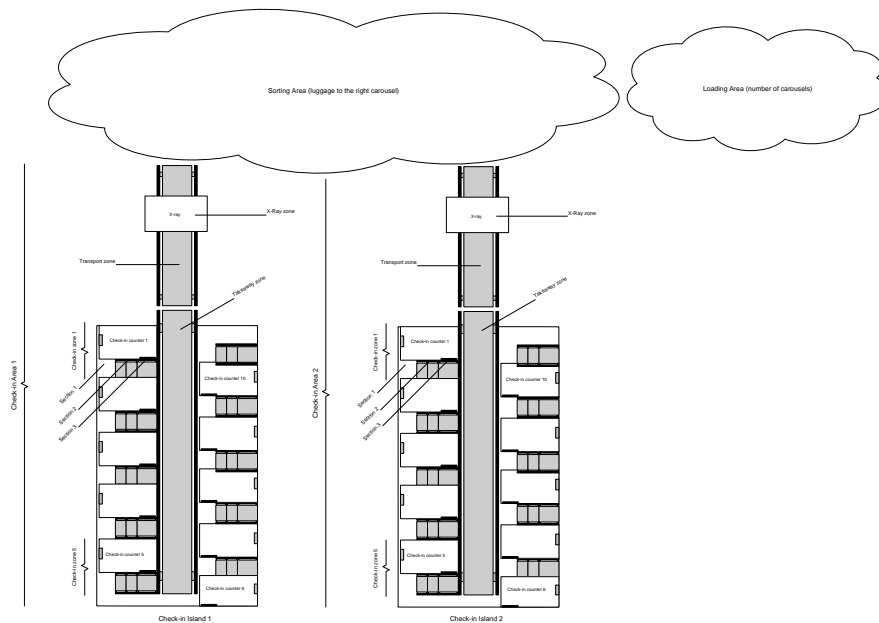


Figure 3.2: Example of the layering structure for the check-in at an airport

The system consists of two check-in islands with a number of check-in counters each, and x-ray machines to scan the luggage. Furthermore, there is a sorting area (with a sorting carousel) and a loading area (with a number of carousels for luggage for different planes). A check-in counter contains three small belts (to transport the luggage onto the takeaway belt), and a number of sensors and motors. This is a pretty basic view of such a system, but it will be sufficient to explain the layer model.

The **system layer** consists of the entire check-in system in this example. It contains components, which handle system functions. An example of such a function is *Starting and stopping of the entire or part of the system*

An **area** is that (geographical) part of a system that fulfils one or more primary system functions. In the check-in example, check-in island 1 (+ x-ray machine), check-in island 2, the sorting carousel, and the loading carousels could be areas<sup>2</sup>.

A **zone** fulfils one system function. In the check-in example, the check-in islands could be split up in a number of check-in zones (the counters), a takeaway zone<sup>3</sup>, a transport zone and an x-ray zone.

<sup>2</sup>All carousels can be different areas or can be combined into one area

<sup>3</sup>Transport luggage from the check-in counters to the 'transport' zone

A **section** is the smallest mechanical/software system building block fulfilling an elementary system function. In the check-in example, a check-in counter (zone) contains sections for each of the three belts. Such a belt section contains all controls (objects) for that belt: motor, and sensors.

**Objects** are electromechanical sensors and actuators installed on the mechanical object, which are necessary to control this mechanical section. Some of the objects are Photo Electric Cells, weight sensors and motors.

### Interfacing

Figure 3.1 already showed some interfacing used by the components (hierarchical or vertical interfacing: report-control, and global interfacing: settings-info). Figure 3.3 shows this interfacing in more detail, including horizontal interfacing (interfacing between sections, zones and areas to handle the product flow through the system).

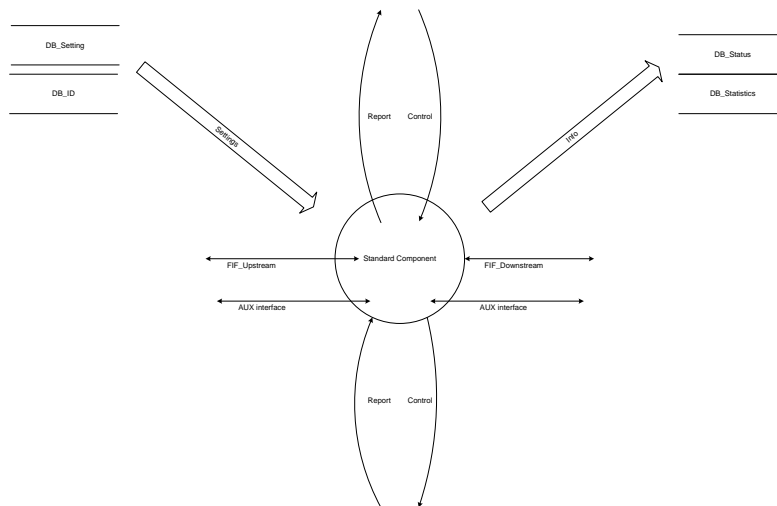


Figure 3.3: Standard component with its standard interfaces

Vertical interfacing is the term used for communication between a manager and its components below. It consists of:

- The control interface, from manager to component(s).
- The report interface, from component(s) to manager.

The second type of interfacing is horizontal interfacing. This type of interfacing is used between the sections, zones and areas and is necessary to handle the product flow through the system. These interfaces include:

- Flow interfaces (FIF\_Upstream, FIF\_Downstream), to hand a physical product from one component to another.
- AUX interfaces, for optional additional information to exchange between components.

The last type of interfacing is global interfacing. Some settings for components influence the system behavior and certain information produced by components needs to be available for the whole system (information that can influence other parts of the system for example). Global interfacing is used for communication with data stores which contain this information and can be accessed by the whole system (all components). This type of interfacing consists of:

- Settings interfaces, to collect settings for the component (from DB\_Settings or DB\_ID).
- Info interfaces, to store information (in DB\_Status or DB\_Statistics) which must be made available for the rest of the system.

### **3.3 Enabling the reuse of components by using blocks**

In order to ease the process of reusing components, the implementation of the reusable components is based on backbone blocks and circle blocks. The meaning of these blocks will be explained in the following paragraphs.

#### **Backbone blocks**

Backbone blocks are components for controlling one or more objects. Usually, these blocks can be found in the object layer of the standard software architecture. There are two types of backbone blocks. Base backbone blocks and add-on backbone blocks. The difference between these two types will be addressed in the circle blocks paragraph below.

#### **Circle blocks**

Circle blocks are components who couple a number of backbone blocks in order to control a complete function. These blocks usually reside on the section level. Circle blocks only control backbone blocks on the same or on a lower level (system, area, zone, section, object) in the architecture. If a slightly different functionality is required, a circle block can easily be adapted by adding or removing backbone blocks to or from it respectively. A circle block always contains a base backbone block. Add-on backbone blocks are optional and add functionality to a base backbone block. The add-on backbone blocks cannot be used stand-alone.

In order to couple the backbone blocks, a circle block provides a backbone interface. The base backbone block functions as a master on the backbone interface. The backbone principle and the data exchange via this backbone will be addressed in the following paragraphs.

#### **Backbone principle**

A circle block provides functionality on the section level by stacking backbone

blocks. The stacked backbone blocks communicate via the backbone interface provided by the circle block. Figure 3.4 shows an example of a basic conveyor circle block.

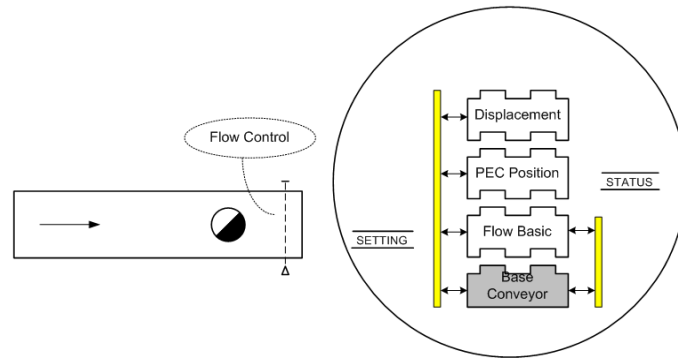


Figure 3.4: Basic conveyor circle block

Here, the circle block constructs the functionality to control a transporting conveyor with flow control.

#### Data exchange via backbone

The backbone interface provided by a circle block is declared in memory bits. These are used as locals by the circle blocks and as globals by the backbone blocks. The interface consists of the following sub interfaces:

- Control interface – This is the interface for signals from the upper level (system, area or zone level).
- Report interface – This is the interface for signals to the upper level.
- Auxiliary interface – This is the interface to auxiliary signals.
- Flow interface – This is the flow interface to adjacent upstream and downstream objects.
- Stack interface – This is the interface where the stacked backbone blocks can store data.
- Transfer stack interface – This is the interface where the stacked backbone blocks for transfers can store data and exchange with a sorter.

### 3.4 Structure components

At Vanderlande, the S7 product package is used to generate the STL code. This package enforces a certain structure of a component (file). This section

addresses this structure. Furthermore, the naming convention for the different types of variables and types of communication will be addressed.

The piece of code shown in Figure 3.5 shows a stripped example of an STL component at Vanderlande:

```
FUNCTION_BLOCK "FB_BA_Flow_Basic"
TITLE =$Revision: 1.1 $CN: 40
//Function:
//...
AUTHOR : MvK
FAMILY : General
NAME : FlowBas
VERSION : 0.0

VAR_INPUT
    i_Var_1 : BOOL := TRUE;
    ...
END_VAR
VAR_OUTPUT
    o_Var_1 : BOOL ;
    ...
END_VAR
VAR_IN_OUT
    io_Var_1 : DWORD ;
    ...
END_VAR
VAR
    s_Var_1 : BOOL ;
    ...
END_VAR
VAR_TEMP
    t_Var_1 : DWORD ;
    ...
END_VAR

BEGIN
NETWORK TITLE =AA: Description 1
    ...
NETWORK TITLE =BA: Description 2
    ...
END_FUNCTION_BLOCK
```

Figure 3.5: Stripped example an STL component

Three main sections can be distinguished in such a component:

- Header information
- Variable declarations
- Actual code

The header information contains information about the type of the component, the name, the family it belongs to and the author. Also, the revision history of the component is logged here. This header is constructed by the S7 package.

The variable declarations are split in (up to) five parts. One for input variables of the component (interface signals, i\_), one for output variables (o\_), one for in- and output variables (io\_) and two for local variables of the component (s\_ and t\_). Usually, not all these parts are used. It depends on the type of variables used in the component of course. The 'instantiations' (input, output and in- and output variables) use the interfacing conventions addressed in the interfacing paragraph in Section 3.2 to communicate with other components.

The actual code of a component can be split in multiple networks. This is to improve the readability of the component. Introduction of a network does not influence code execution however. In Figure 3.5, network AA ‘falls through’ to network BA for example (unless an explicit jump to somewhere else is introduced in network AA).

Besides the variables declared in the variable declarations, global memory can be addressed directly as well. This goes in the form of so-called ‘merkers’. Merkers are memory locations (bits, words, doubles) in a PLC which can be read and written by all components on the PLC. These merkers conform to the same interfacing conventions as instantiations (i., o., io.). This means there are special chunks of memory available for the different types of interfaces.

### 3.5 Structure STL instructions

In this section, a number of constructions and operators of STL are addressed (in more detail) and illustrated by examples.

#### Header information

Figure 3.6 shows an example of the header information of a component in detail.

```

FUNCTION_BLOCK "FB_BA_Flow_Basic"
TITLE =$Revision: 1.1 $CN: 40
//Function:
//Backbone block for basic flow control
//
//Description:
//The basic flow control consists of the functions gap control and head to head\
//control.
//
//
//      +-----+          +-----+
// --> | 2 |          | 1 | --> flow direction
//      +-----+          +-----+
//          |<- gap ->|
//          |<- head to head ->|
//
//History:
//$Log: FB_BA_Flow_Basic.AWL,v $
//Revision 1.1 2004-02-04 16:59:26+01 nlgm
//Several modifications during SSS implementation phase 'B'.
//
//Revision 1.0 2003-12-24 12:15:26+01 nlgm
//Initial revision
//
//
AUTHOR : MvK
FAMILY : General
NAME : FlowBas
VERSION : 0.0
...
...

```

Figure 3.6: Header information of FB\_BA\_FB\_Basic

Most of this information is presented as comment (behind ‘//’). As can be seen, the component name (FB\_BA\_Flow\_Basic) and component type (FUNCTION\_BLOCK) can be found on the first line. The amount of revisions (1) can be extracted from the second line and the component family (General) from the line starting with ‘FAMILY’. Date and time of a given revision as well as the programmer/submitter of that revision can be found in the lines starting with

//Revision.

**Decisions**

STL has numerous instructions that are (part of) a decision. These instructions include boolean operations (BoolOp, TerBoolOp), mathematical operations (MathOp), numerous types of jumps (JumpOp) and a loop-instruction (LoopOp).

BoolOp	A, AD, AN, AW, O, OD, ON, OW, X, XN, XOD, XOW, A(), AN(), O(), ON(), X(), XN()
TerBoolOp	S, R, =

Table 3.1: Boolean operators in STL

The boolean operations are summed up in Table 3.1. These operations are used for logic operations on signals, and to set certain bits (A, AN, O, ON, X, XN, A(), AN(), O(), ON(), X(), and XN()), words (AW, OW, and XOW), or doubles (AD, OD, and XOD) in memory.

The operations work with bits in the status word of a PLC. This status word contains bits that can store parts of a calculation or boolean operation. The most important bits are called the first check ( $\overline{FC}$ ), and result of logic operation (RLO) bits. If the  $\overline{FC}$  bit is 0, this indicates that the next boolean operation begins a new boolean string. In this case, the signal state (parameter of that boolean operation) is stored directly in the RLO bit (and the  $\overline{FC}$ -bit will be set to 1).

If the  $\overline{FC}$ -bit is 1, the RLO bit is used directly as (one of the two) input(s) of the boolean operation (the other input is the signal state). The result of the boolean operation is stored in the RLO bit again. The boolean operations often lead to if-then and if-then-else constructs in STL code as can be seen in the following example.

```
// If-then/If-then-else example
FA01: A      #i_inst1; // If the signal state at input #0_inst1 is 1.
           // (FC, and RLO initially 0, #0_inst1 is 1 ->
           // RLO=1, FC=1)
      AN     #i_inst2; // AND the signal state at input #0_inst2 is 0.
           // (FC=1, RLO=1, #0_inst2 is 0 -> RLO=1)
      =      #io_inst3; // THEN set #io_inst3 to 1. ELSE set #io_inst3 to 0.
           // (#io_inst3=1 (or 0), FC=0, RLO=0)
```

The mathematical- and jump operations are summed up in Table 3.2.

MathOp	==I, <>I, <I, <=I, >I, >=I, ==D, <>D, <D, <=D, >D, >=D, ==R, <>R, <R, <=R, >R, >=R
JumpOp	JZ, JNBI, JBI, JM, JMZ, JN, JOS, JO, JP, JPZ, JCN, JNB, JC, JCB, JUO

Table 3.2: Mathematical- and jump operators in STL

These operations are often used together in comparing values and jumping accordingly in the code. Usage of some mathematical and jump operations is shown in the following example.

```

// Jump example
FA03: L    10; // Load integer 10 into accumulator 1 low as lower limit.
      L    MW11; // Load the value in merker word MW11 into accumulator 1 low,
           // transferring the integer value 10 into accumulator 2 low.
      <=I    // Is 10 less than or equal to the value in MW11? If so, then set
           // Result of Logic Operation (RLO) to 1; otherwise reset the RLO to 0.
      JC    FA04; // IF RLO = 1 THEN jump to FA04.
      ...    // ELSE perform instructions
      ...    // on these lines.
FA04: ...

```

Finally, there is the LOOP operator used for loops.

LoopOp	LOOP
--------	------

Table 3.3: LOOP operator in STL

An example of the usage of a loop operator is shown in the following example.

```

// Loop example
      L    +5; // Initialize Loop Counter.
FA03: T    MB10; // AND the signal state at input #0_inst2 is 1.
      ...    // Do something
      L    MB10;
      LOOP FA03; // Decrease loop counter (in accumulator) and
           // return to FA03 if the counter is higher than 0.

```

For more details on all operators, see the Simatic S7 reference manual [Sie98].

## Calls

In STL, there are three types of calls and they can be to functions (1 type) and to function blocks (2 types). A call to a function has the following form:

```
CALL "FC_Make_CS" (
```

Where FC\_Make\_CS is the function that is being called. For a ‘standard’ call to a function block, it is slightly more difficult to find the physical name of the function block that is being called. The following call is to a function block:

```
CALL #s.FB_Displacement (
```

The name in the call-line (s.FB\_Displacement) is a local name for function block (FB\_BA\_Displacement). Using this local name, the physical name (FB\_BA\_Displacement) can be extracted from the variable declaration at the beginning of the STL-file however.

```
s_FB_Displacement : "FB_BA_Displacement";
```

The second type of call to a function block is by making use of a data instance block:

```
CALL "FB_Area" , "DI_Area" (
```

Data instance blocks contain information or parameters that can be used during execution of the function block called.

## Naming convention merkers at Vanderlande

The naming convention for merkers in the General library at Vanderlande consists of:

M\_SI.<unique name> and MW\_SI.<unique name> for Stack Interfaces  
M\_R.<unique name> and MW\_R.<unique name> for Reporting  
M\_C.<unique name> and MW\_C.<unique name> for Controlling  
M\_AUX.<unique name> and MW\_AUX.<unique name> for Auxiliary  
M\_FU.<unique name> and MW\_FU.<unique name> for Flow interfaces Up-  
stream  
M\_FD.<unique name> and MW\_FD.<unique name> for Flow interfaces Down-  
stream

The Warehouse and Distribution (W&D) library can use slightly different names for the flow- and stack interfaces:

M\_FIF\_1.<unique name> and MW\_FIF\_1.<unique name> for Flow Interface transfer Side 1  
M\_FIF\_2.<unique name> and MW\_FIF\_2.<unique name> for Flow Interface transfer Side 2  
M\_TSI.<unique name>, MW\_TSI.<unique name>, MD\_TSI.Location and MD\_TSI.Location.2 for Transfer stack interface

Using this naming convention, it is possible to identify a merker in a component. merkers can occur on every location where a memory location can be used in STL (behind BoolOps and TerBoolOps).

### **Naming convention instances at Vanderlande**

The naming convention for instances at Vanderlande consists of:

#i.<unique name> for input instances in a given component  
#o.<unique name> for output instances in a given component  
#io.<unique name> for in- and output instances in a given component

As with merkers, instantiations can occur on every location where a memory location can be used in STL.

## **3.6 Metrics to use for STL**

As stated in Section 2.4, it would therefore be useful if both McCabe and information flow metrics would be collected for STL components. The structures used in decisions and in the interaction of components are known. This makes the collection of these metrics possible.

Lines of code are far more easy to count/collect however, so this metric could still be a good starting point for the collection process. The same holds for amount of comment (or comment ratio) and amount of revisions.

Finally, a number of metrics on revisions (amount of revisions for a component, date+time, amount of coders, etcetera) can be retrieved from the header information.

## **3.7 Availability of tools who can be used for the collection of metrics**

As stated in Section 2.2, basically nothing has been done in the field of metrics on STL. Therefore there are no specific tools available to collect metrics for

STL either. With Step7 it is possible to retrieve the size of a single component (in bytes), but since this basically is the only metric it can collect, it is rather limited for our purpose.

Therefore, the suggestion was made to write a simple parser for this project in order to collect the necessary metrics from the .awl-files. By doing this it will also be possible to build prototypes in an iterative way. These can be used to show possibilities of metrics to collect to Vanderlande. This delivers feedback and helps in defining the requirements for the complexity metrics set. The tool or parser written (AWL Analyzer) will be explained in detail in Section 4.

## 4 AWL Analyzer

AWL Analyzer is a tool to analyze and visualize (a collection of) Anweisungs Liste (.awl) files. It was developed iteratively via rapid prototyping. The tool extracts numerous metrics from .awl-files, stores them in a database, and returns this information in the form of sortable tables. Furthermore, the three types of communication between .awl-components (calling, merker interaction, and instantiation interaction) are visualized in a selection of graphs. The tables on the one side and the graphs on the other side form the two main sections of the tool. The way AWL Analyzer scans and parses STL code, the collection (and presentation) of the metrics, and the graph representations will be addressed in the sections below.

### 4.1 Scanning and parsing STL

Since STL contains (a maximum of) one statement per line, the decision was made to scan an .awl-file line per line. All .awl-files selected are therefore stored line per line in a list of strings. For all tables in the database (metrics, calls, merkers, instantiations, and revisions), this list is processed once. This way, the AWL Analyzer can be constructed in a more component wise way.

AWL Analyzer is developed in Java, and thus uses Java constructs to analyze each line for occurrences of strings and characters. The constructs used include:

- **line.startsWith("<this string>");** To check if string 'line' starts with string "<this string>" (at position 0). TRUE if so, FALSE otherwise.
- **line.startsWith("<this string>", x);** To check if string 'line' starts with string "<this string>" at position x. TRUE if so, FALSE otherwise.
- **line.indexOf("<this string>");** To find the index of the first occurrence of "<this string>" in 'line'. Integer with position of occurrence if 'line' contains "<this string>", -1 otherwise.
- **line.charAt(i);** Return the character at position i of string 'line'.
- **line.substring(y, z);** Return the string between position y and z of string 'line'.

As explained in Section 3.4, S7 enforces or creates a certain layout for STL files. This holds for the header information as well as for the variable declarations and the actual code. Component name, component type, revision number and component family in the header information always start at a fixed position of a line.

In the actual code, the standard layout (statements always start at position 6 and addresses at position 12 of a line) is used to detect calls, merker operations, and instantiation operations. Occasionally, merkers and instantiations start at position 36 of a line.

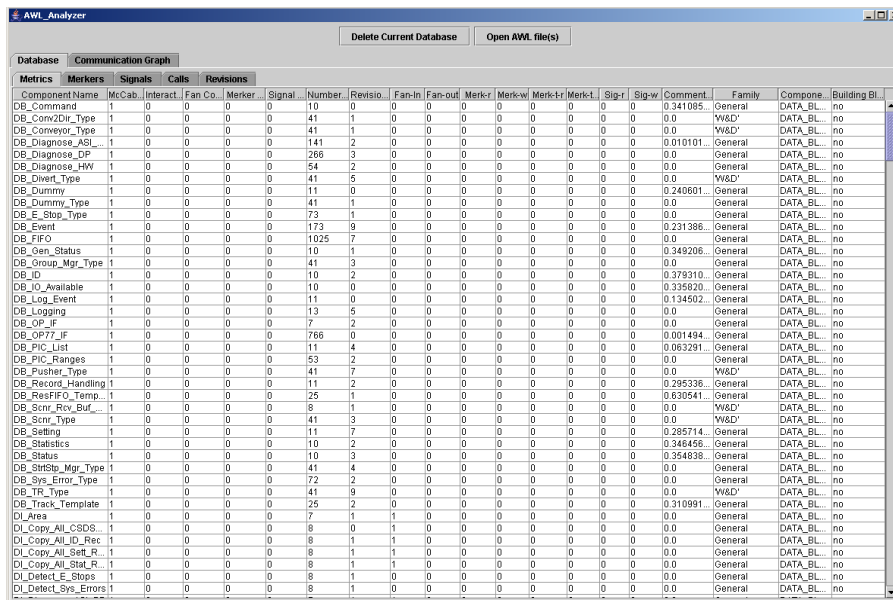
This means that most detection or line analysis is performed with the startsWith commands. It would be possible (and even easy) to perform the analysis with

indexOf commands, but at the moment this is useless. With indexOf commands, no extra calls, merker operations, and instantiation operations will be found (except from operations behind comment slashes “//”).

If a line starts with comment slashes “//”, this means that an eventual occurrence of a call, merker, or instantiation in that line should not be counted. In combination with the forced layout structure by STL, this check guarantees that all calls, merkers, and instantiations are detected, but only if they occur outside the comment.

## 4.2 Collecting metrics from .awl-files

Figure 4.1 shows a capture of AWL Analyzer in the database/tables view after opening a library of .awl-files<sup>4</sup>.



Component Name	McCab	Interact	Fan Co	Merker	Signal	Number	Revisio	Fan-In	Fan-out	Merker	Merkerw	Merker	Merker	Sign	Sign	Comment	Family	Compon	Building Bl
DB_Command	1	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0.341065..	General	DATA.BL..	no
DB_Corv2Dir_Type	1	0	0	0	0	41	1	0	0	0	0	0	0	0	0	0.0	W&D'	DATA.BL..	no
DB_Corveyor_Type	1	0	0	0	0	41	1	0	0	0	0	0	0	0	0	0.0	W&D'	DATA.BL..	no
DB_Diagnose_ASI...	1	0	0	0	0	141	2	0	0	0	0	0	0	0	0	0.010101..	General	DATA.BL..	no
DB_Diagnose_DP	1	0	0	0	0	266	3	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_Diagnose_HW	1	0	0	0	0	54	2	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_Divert_Type	1	0	0	0	0	41	5	0	0	0	0	0	0	0	0	0.0	W&D'	DATA.BL..	no
DB_Dummy	1	0	0	0	0	11	0	0	0	0	0	0	0	0	0	0.240601..	General	DATA.BL..	no
DB_Dummy_Type	1	0	0	0	0	41	1	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_E_Stop_Type	1	0	0	0	0	73	1	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_Event	1	0	0	0	0	173	9	0	0	0	0	0	0	0	0	0.231386..	General	DATA.BL..	no
DB_FIFO	1	0	0	0	0	1025	7	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_Cen_Status	1	0	0	0	0	10	1	0	0	0	0	0	0	0	0	0.349206..	General	DATA.BL..	no
DB_Group_Mgr_Type	1	0	0	0	0	41	3	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_ID	1	0	0	0	0	10	2	0	0	0	0	0	0	0	0	0.379310..	General	DATA.BL..	no
DB_IO_Available	1	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0.335820..	General	DATA.BL..	no
DB_Log_Event	1	0	0	0	0	11	0	0	0	0	0	0	0	0	0	0.134502..	General	DATA.BL..	no
DB_Logging	1	0	0	0	0	13	5	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_OP_IF	1	0	0	0	0	7	2	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_OP77_IF	1	0	0	0	0	766	0	0	0	0	0	0	0	0	0	0.001494..	General	DATA.BL..	no
DB_PIC_List	1	0	0	0	0	11	4	0	0	0	0	0	0	0	0	0.063291..	General	DATA.BL..	no
DB_PIC_Ranges	1	0	0	0	0	53	2	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_Pusher_Type	1	0	0	0	0	41	7	0	0	0	0	0	0	0	0	0.0	W&D'	DATA.BL..	no
DB_Record_Handling	1	0	0	0	0	11	2	0	0	0	0	0	0	0	0	0.295336..	General	DATA.BL..	no
DB_ResFIFO_Temp	1	0	0	0	0	25	1	0	0	0	0	0	0	0	0	0.630541..	General	DATA.BL..	no
DB_Schr_Rcv_Buf...	1	0	0	0	0	8	1	0	0	0	0	0	0	0	0	0.0	W&D'	DATA.BL..	no
DB_Schr_Type	1	0	0	0	0	41	3	0	0	0	0	0	0	0	0	0.0	W&D'	DATA.BL..	no
DB_Setting	1	0	0	0	0	11	7	0	0	0	0	0	0	0	0	0.285714..	General	DATA.BL..	no
DB_Statistics	1	0	0	0	0	10	2	0	0	0	0	0	0	0	0	0.348456..	General	DATA.BL..	no
DB_Status	1	0	0	0	0	10	3	0	0	0	0	0	0	0	0	0.354838..	General	DATA.BL..	no
DB_StrtStp_Mgr_Type	1	0	0	0	0	41	4	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_Sys_Error_Type	1	0	0	0	0	72	2	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DB_TR_Type	1	0	0	0	0	41	9	0	0	0	0	0	0	0	0	0.0	W&D'	DATA.BL..	no
DB_Track_Template	1	0	0	0	0	25	2	0	0	0	0	0	0	0	0	0.310991..	General	DATA.BL..	no
DI_Area	1	0	0	0	0	7	1	1	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DI_Copy_All_CSOS...	1	0	0	0	0	8	0	1	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DI_Copy_All_ID_Res	1	0	0	0	0	9	1	1	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DI_Copy_All_Sett_R...	1	0	0	0	0	8	1	1	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DI_Copy_All_Sett_R...	1	0	0	0	0	8	1	1	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DI_Detect_E_Stops	1	0	0	0	0	8	1	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no
DI_Detect_Sys_Errors	1	0	0	0	0	8	1	0	0	0	0	0	0	0	0	0.0	General	DATA.BL..	no

Figure 4.1: Metrics table after opening a collection of .awl files

As can be seen in Figure 4.1, the database section of the tool consists of five tables: metrics, markers, instantiations, calls, and revisions.

The metrics table contains a number of metrics for each component. The markers, instantiations and calls tables contain information about the interaction between the components (all data exchanges are logged here). The revisions table contains information on the revisions of the components (extracted from the header information of the components).

<sup>4</sup>It is possible to open a single .awl-file or a whole library of .awl-files via ‘Open AWL file(s)’. An entry is added to the database/table(s) for every distinct component, call, etcetera in these files.

The actual detection, collection and construction of all metrics and information presented in the tables will be addressed in the following sections. The first section will cover the non-interaction metrics from the metrics table as well as the contents of the revisions table. The second section is dedicated to the interaction metrics from the metrics table as well as the merker-, instance- and call tables.

Finally, a small section will be dedicated to the sorting functionality integrated in the tables.

#### 4.2.1 Non-interaction metrics and revision detection

This section contains details on how to detect, collect and construct the non-interaction metrics from the metrics table of AWL Analyzer as well as the contents of the revision table.

##### **Component Name, Type, Family and Revisions**

These metrics and pieces of information can be collected from the header information of a component as can be seen in Section 3.5. Step7 constructs and lays out the structure for the header information of a component for a big part. This makes it relatively easy to collect those metrics.

Component name, component type, and component family are stored in the metrics table of the database for every component found. Component name, revision number, date+time revision, and submitter are stored in the revision table of the database for all revisions found.

Information like component type and component family can be useful for making selections based on subsets of the code library. This is the reason they are collected.

##### **Number of lines of code and comment ratio**

As stated in Section 2.3, there are numerous ways to count the number of lines of code and the comment ratio. For the amount of lines, the choice was made to include comment lines, but to exclude totally empty lines. For the comment ratio, the amount of characters behind comment slashes ('//') is divided by the amount of characters before comment slashes. This gives the highest accuracy for comment ratio.

##### **McCabe complexity (Decision points)**

In order to count decision points in STL, a definition is needed for what we count as a decision point in STL.

Each of the jump operations and the loop operation addressed in Section 3.5 form a decision point on themselves and thus should be counted. There is a slight dilemma with the BoolOp, TerBoolOp and MathOp instructions however. They all form part of 'if then/if then else' constructions in order to check and set (controlling) bits, but they do not really influence the information flow in a component. They only influence the information flow in combination with jump or loop operations who are already counted.

This is the reason the decision was made to only count the amount of JumpOps and LoopOps in a component. As stated in Section 2.3, the McCabe (or cyclomatic) complexity is the amount of decision points + 1. In pseudo code:

```
Decide_Count = 0;
for line 1 to last line of component
  if the line contains an operator (JumpOp/LoopOp) then
    Decide_Count = Decide_Count + 1;
  fi
rof
McCabe Complexity = Decide_Count + 1;
```

Of course, only operations before comment slashes ‘//’ on a line should be counted.

#### 4.2.2 Interaction detection

Calls, merker- and instantiation operations form the interaction between STL components. In AWL Analyzer, these three types of operations are logged in separate tables of a database. Furthermore, a number of metrics in the metrics-table are constructed based on the tables with these three types. The collection and construction of the call table, merker table and instantiation table including related metrics will be addressed in the sections below. Finally, all interaction metrics will be combined in the interaction complexity section.

##### Call table and fanIO metrics

Section 3.5 mentioned the structures and occurrences of calls in STL. Based on these structures, it is possible to collect a number of data items for a call.

For every call, an entry is stored in the call-table of the database. This table contains columns for:

- Component: Component name of the calling component
- Comp Fam: Component family
- Calls: Component name of the component called
- Call Fam: Family of component called
- Data Instance: Data instance used in the call if available<sup>5</sup>

Based on this information, fan-in, fan-out and fan-complexity (fan-in\*fan-out) can be calculated as well.

##### Merker table and MerkerIO metrics

As stated in Section 3.4, merkers are common memory locations. They are used by components as a way of communicating with the rest of the PLC system (components, managers). The memory locations can be single bits, words or

---

<sup>5</sup>If no data instance is used, this entry contains a “-”.

doubles<sup>6</sup>. Section 3.5 already addressed the identification of merkers (naming convention and occurrence).

Furthermore, two types of operation can be performed on a merker, namely reading and writing. Based on the STL manual [Sie98] and the STL libraries available, the following could be concluded on the operations:

If a BoolOp (see Table 3.1 in Section 3.5) is performed on a merker or if the merker is Loaded from memory ‘L merker’, a reading operation takes place. If a TerBoolOp is performed on a merker or if the merker is Transferred to the memory (‘T merker’), a write operation takes place.

Based on the identification of merkers and the type of operation on them, the merker table can be constructed. This table contains an entry for every (component, merker, operation)-combination in the components analyzed. There are columns for component name, merker name, type of operation and amount of identical operations.

Based on this table, distinct merker reads, distinct merker writes, total merker reads, total merker writes and merker complexity (distinct merker reads \* distinct merker writes) can be constructed for a given component.

#### **Instantiation table and InstantiationIO metrics**

As stated in Section 3.4, instances are very similar to merkers. The occurrence and detection are pretty much the same. As can be seen in Section 3.5, the difference resides in the names of the variables (the naming convention).

Formally, input instances (#i\_) should only be read and output instances (#o\_) should only be written in a component. In practice however, this sometimes is not the case. This led to the decision to use the same procedures for operation detection (read/write) as for the merkers. The instantiation table is similar to the merker table: component name, instantiation name, type of operation and amount of identical operations.

The calculation of distinct instantiation reads, distinct instantiation writes and instantiation complexity can be done based on the information in this table.

#### **Interaction complexity**

Fan-in, fan-out, merkers read, merkers written, instances read and instances written can be extracted and counted for a given component. This gives the input for an ‘interaction metric’. The question is how to form an overall metric based on these separate metrics however.

Based on numerous conversations and discussions on this issue with the PLC programmers, it was concluded that merker reads and merker writes were more important than calls and therefore should have a higher impact on the overall metrics set. Furthermore, instances could be treated the same way as merkers. An instance read/write had approximately the same impact as a merker read/write according to the PLC programmers.

---

<sup>6</sup>The double merkers/signals are only used virtually in order to identify what memory intervals are used for what purposes. They are thus not important to track or count here and are therefore not addressed.

The IEEE 982.2 formula for informational complexity (IFC) is [IEEE88]:

**IFC = (fanin \* fanout)<sup>2</sup>** with:

fanin = local flows into a procedure + number of data structures from which the procedure retrieves data  
fanout = local flows from a procedure + number of data structures that the procedure updates

IFC basically represents the number of paths that can go through the procedure/function/component in question (or at least an upper bound for that).

Initially, the idea was to alter this formula in order to incorporate the difference in importance between calls and merkers/instances, but later we discovered that the difference in importance was already integrated in the formula. On average, there are a lot more merkers operations than calls in a component.

This initiated the decision to use this formula with only a slight moderation. The square was eliminated, since this basically only multiplies the figure and delivers higher numbers. The formula used for calculating the interaction complexity (IC) of a component therefore became:

$$\text{IC} = (\text{fan-in} + \text{merker-reads} + \text{instance-reads}) * (\text{fan-out} + \text{merker-writes} + \text{instance-writes})$$

### 4.2.3 Sorting tables

All tables have sorting functionality and the possibility to swap columns. This can be done by clicking (and dragging) column headers. This makes it easier to find and analyze specific components. For example based on component name, McCabe complexity or interaction complexity. An example of a library of components sorted on McCabe complexity (descending) is shown in Figure 4.2.

Component Name	Mc	Interact	Fan Co	Merker	Signal	Number	Revisio	Fan-in	Fan-out	Merker	Merkw	Merckr	Merkt	Sig-r	Sig-w	Comment	Family	Compose	Building B
FB_BA_Update_Track	60	483	13	14	104	1333	-1	1	13	7	2	58	2	13	8	1.297931...	General	FUNCTION	no
FB_OP_Controller	53	12	0	0	3	1261	14	0	3	0	0	0	0	3	11	1.400299...	General	FUNCTION	no
FB_OF77_Controller	51	24	0	0	15	1434	0	0	3	0	0	0	0	3	5	1.865459...	General	FUNCTION	no
FC_Record_Index	41	0	0	0	0	895	3	0	0	0	0	0	0	0	0	1.355026...	General	FUNCTION	no
FB_BA_PEC_Position	37	672	15	221	20	759	16	15	1	17	13	36	19	10	2	1.534489...	General	FUNCTION	no
FB_Diagnose_ASI_FF	31	40	2	0	21	899	6	1	2	0	0	0	0	7	3	0.874471...	General	FUNCTION	no
FB_BA_PFI_Displacement	31	160	0	49	7	476	13	6	0	7	7	18	11	7	1	1.214599...	General	FUNCTION	no
FC_FIFO	30	9	0	0	3	558	19	0	2	0	0	0	0	3	1	1.194444...	General	FUNCTION	no
FB_BA_PEC_Trigger	30	228	16	24	30	554	17	16	1	12	2	13	2	10	3	1.704271...	General	FUNCTION	no
FB_DP_Sp_Sens_Rev	28	6	0	0	6	562	18	0	0	0	0	0	0	2	3	0.902519...	General	FUNCTION	no
FB_Make_OS	27	0	0	0	0	451	1	0	0	0	0	0	0	2	0	1.004622...	General	FUNCTION	no
FB_BA_Induct_Seq_2	24	560	0	195	57	540	10	1	0	15	13	28	15	19	3	1.408807...	W6D	FUNCTION	no
FB_BA_Induct_Seq_1	24	592	0	209	57	541	10	2	0	16	13	29	15	19	3	1.415742...	W6D	FUNCTION	no
FB_BA_Displacement	22	160	0	24	0	271	6	23	0	4	6	11	10	3	0	1.072391...	General	FUNCTION	no
FC_Logging	20	24	3	0	5	375	7	1	3	0	0	0	0	5	1	1.770528...	General	FUNCTION	no
FC_Any_ASCII_To_Dirt	19	0	0	0	0	460	11	0	1	0	0	0	0	0	0	1.308495...	General	FUNCTION	no
FB_BB_Scanner	19	476	6	49	144	451	5	3	2	7	7	12	8	18	8	1.153023...	W6D	FUNCTION	no
FB_Induct_On_ResLink	18	462	0	109	26	466	15	0	7	9	12	20	13	13	2	1.340452...	General	FUNCTION	no
FB_Covw_Records	18	7	4	0	0	460	3	4	1	0	0	0	0	3	0	1.188508...	General	FUNCTION	no
FB_BB_Divert_Pusher	17	920	0	270	152	409	9	3	0	18	15	44	17	19	8	1.208688...	W6D	FUNCTION	no
FC_Winb_Track_Unc	15	0	0	0	0	219	4	12	0	0	0	0	0	3	0	2.172394...	General	FUNCTION	no
FC_Any_ASCII_To_Int	15	2	1	0	0	380	9	1	1	0	0	0	0	1	1	1.200269...	General	FUNCTION	no
FB_Mailbox_ASI_FF	15	80	3	0	49	314	5	11	3	0	0	0	0	7	7	0.931013...	General	FUNCTION	no
FB_Gen_Win_FIFo_Link	15	24	0	0	0	345	7	0	3	0	0	0	0	8	0	1.768092...	General	FUNCTION	no
FB_BB_Covw_2dir	15	4200	0	2310	200	882	32	4	0	55	42	116	46	25	8	1.576158...	General	FUNCTION	no
FB_BA_Divert_Seq_2	15	432	0	224	20	380	11	1	0	16	14	24	16	10	2	1.448427...	W6D	FUNCTION	no
FB_BA_Divert_Seq_1	15	496	0	224	20	380	9	5	0	16	14	24	16	10	2	1.458630...	W6D	FUNCTION	no
FC_Int_To_Any_ASCII	14	2	0	0	1	363	11	0	1	0	0	0	0	1	1	1.268866...	General	FUNCTION	no
FC_Dirt_To_Any_ASCII	14	2	0	0	1	406	11	0	1	0	0	0	0	1	1	1.226060...	General	FUNCTION	no
FB_SlurpMan_Manager	14	72	0	0	72	496	13	0	0	0	0	0	0	12	6	1.348962...	General	FUNCTION	no
FB_Induct_On_Resense	14	340	0	109	22	376	13	0	3	9	12	20	13	11	2	1.352229...	General	FUNCTION	no
FB_Diagnose_DP	14	12	4	0	0	294	2	1	4	0	0	0	0	2	0	1.375274...	General	FUNCTION	no
FB_System_Facilities	13	3	3	0	0	397	10	1	3	0	0	0	0	0	0	1.203510...	General	FUNCTION	no
FB_Gen_Win_Flow_FIFo	13	21	0	0	0	323	5	0	3	0	0	0	0	7	0	1.700942...	General	FUNCTION	no
FB_BA_PEC_Stop	13	210	0	63	30	214	7	2	0	8	7	10	7	10	7	1.829787...	General	FUNCTION	no
FB_AG_LSEND_LRCV	13	5	0	0	2	300	6	0	3	0	0	0	0	1	2	2.340380...	General	FUNCTION	no
FB_Gen_Win_Prg_A	12	32	0	0	8	229	5	0	3	0	0	0	0	8	1	1.610576...	General	FUNCTION	no
FB_BA_PEC_Fill_Level	12	80	0	18	14	198	10	3	0	6	3	10	3	7	2	2.530957...	General	FUNCTION	no
FB_BA_Mult_Spd	12	70	0	0	45	163	1	3	0	2	0	3	0	9	5	0.449242...	General	FUNCTION	no
FB_AGSSEND_AGRCV	12	0	0	0	0	265	4	0	2	0	0	0	0	2	1	1.971094...	General	FUNCTION	no
FC_PIC_Generator	11	16	9	0	1	249	10	3	3	0	0	0	0	1	1	1.108466...	General	FUNCTION	no

Figure 4.2: Metrics sorted on McCabe complexity in descending order

### 4.3 Representing communication structure in graphs

The communication graph tabular is one of the two main sections in AWL Analyzer. It can be used to visualize call, merker, and instantiation graphs. Furthermore, numerous filtering actions can be performed on those graphs in order to increase their understandability. Information to construct the graph is retrieved from the metrics database. The construction of this database was addressed in Section 4.2. Standard graph drawing and available options to filter the graph for increased understandability will be addressed in the following sections.

#### 4.3.1 Standard graph drawing

The communication graph tabular consists of two main sections. The graph control panel on the left and the graph itself on the right. The graph control panel consists of:

- Checkbox ‘Eliminate Individuals’: if this checkbox is checked, individual nodes (no incoming and outgoing edges) are filtered out.
- Radiobuttongroup ‘Edge/Node Selection’: what edges and nodes to draw. At the moment, there are 5 options:
  - MerkerRead: Draw all components reading merkers (rectangles) and the merkers read (purple diamonds) as nodes and draw an arrow (edge) from a merker to a component if this merker is read by the

component. Figure 4.3 shows an example of (part of) such a MerkerRead graph for an .awl-library.

- MerkerWrite: Draw all components writing markers (rectangles) and all markers written by components (purple diamonds) as nodes and draw an arrow (edge) from a component to a marker if this component writes to this marker.
- MerkerReadWrite: Draw both MerkerReads and MerkerWrites in one graph. The reason for separate options for MerkerRead, MerkerWrite and MerkerReadWrite is performance. This issue is addressed in a separate section.
- Instance: Draw the interaction between components (squares) and signals (purple diamonds). An arrow from a component to an instance if the instance is written by the component. An arrow from an instance to a component if the instance is read by the component.
- Call: Draw the interaction (calls) between components.

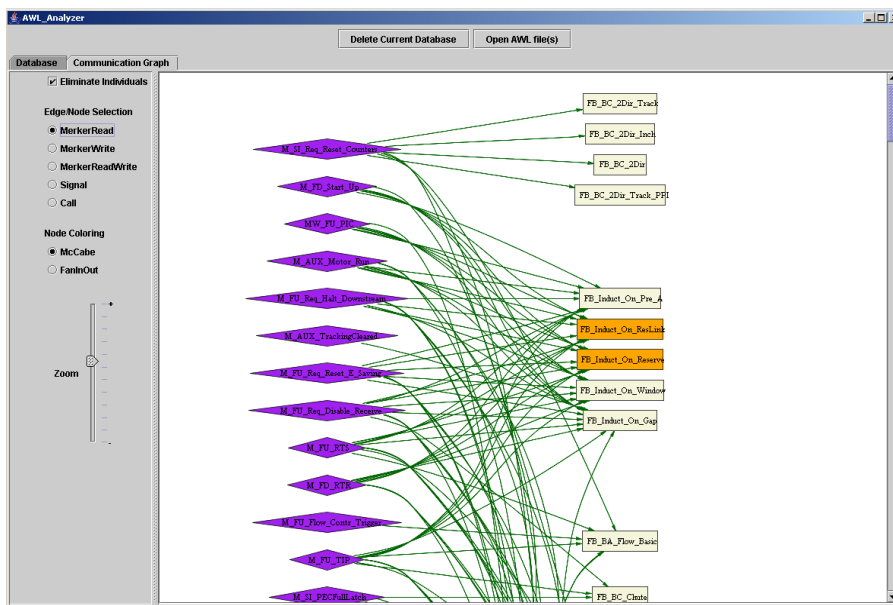


Figure 4.3: MerkerRead graph, McCabe complexity node coloring

- Radiobuttongroup 'Node coloring': Based on the type of complexity, a different coloring for the nodes is used. At this moment, there are 2 options:
  - McCabe: Color nodes based on their McCabe complexity. There are 4 levels of coloring.
    - \* McCabe complexity 0-10: Beige (background color) or grey (if this component is not a member of the .awl-selection. Standard Siemens components for example).
    - \* McCabe complexity 11-20: Orange

- \* McCabe complexity 21-50: Red
- \* McCabe complexity >50: Dark Red
- FanInOut: Color nodes based on their FanInOut complexity. There are 4 levels of coloring.
  - \* FanInOut complexity 0-5: Beige (background color) or grey (if this component is not a member of the .awl-selection. Standard Siemens components for example).
  - \* FanInOut complexity 6-10: Orange
  - \* FanInOut complexity 11-20: Red
  - \* FanInOut complexity >20: Dark Red

Figure 4.4 shows an example of (part of) a callgraph using FanIn-Out coloring. One can see that components FC\_PIC\_Generator, FC\_Write\_BOOL\_Record and FC\_Log\_Event\_Track have a FanInOut complexity between 6 and 10 and that FC\_Log\_Event has a FanInOut complexity of higher than 20.

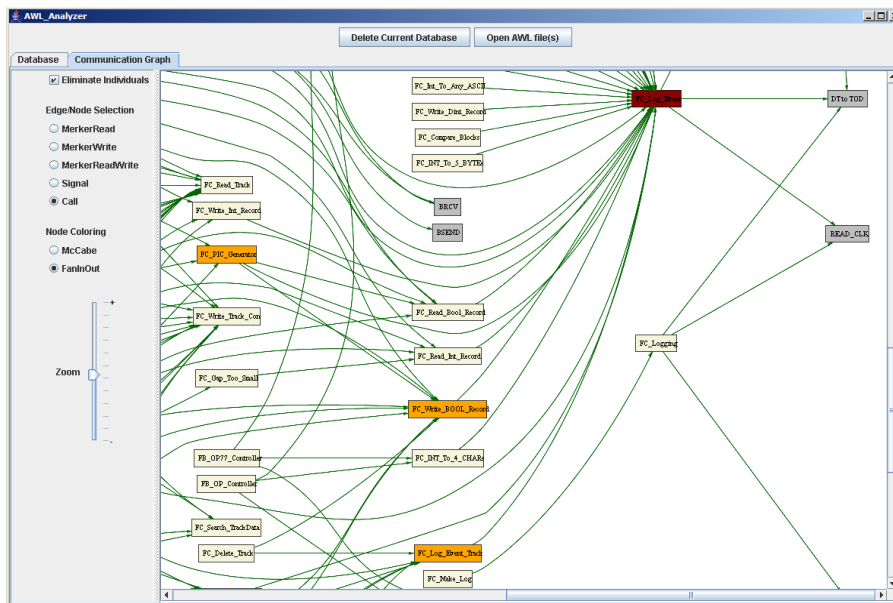


Figure 4.4: Call graph, FanInOut complexity node coloring

- Zoom slider: To zoom in or out in the graph.

### 4.3.2 Performance problems for graph drawing

There are separate options (graphs) for MerkerRead, MerkerWrite, and MerkerReadWrite. As stated before, this is due to performance issues. Calculating the layout for the MerkerReadWrite graph of a complete .awl-library can take up to a couple of minutes on a 2.66 GHz Intel Pentium. This is due to the use

of ‘dot’ as layout algorithm. Dot delivers the clearest and most neat graphs from the graph layout algorithms we considered. Since clarity of the graph was considered more important than performance, the dot algorithm was chosen.

Applying a layout algorithm to the instantiation graph has the same performance problems as applying the algorithm to the MerkerReadWrite graph. This impact is far less than for the MerkerReadWrite graph however. It takes 20 seconds or less to apply the algorithm to the instantiation graph of a whole library. This means no separate graphs are needed for instantiation reads and instantiation writes.

### 4.3.3 Coloring dependencies and drawing subgraphs

The layout algorithm used by AWL Analyzer (Dot) in general produces neat graphs. Sometimes however it is still hard to follow the information flow through the graph. To help in coping with this matter, two options are built in. The first option consists of coloring the direct neighbors<sup>7</sup> of a node when clicking it. Components calling (or merkers/instances read by) the clicked component are colored blue. Components called (or merkers/instances written) by the component are colored yellow. Figure 4.5 shows an example where node FB\_BA\_ShiftUpd\_Track is clicked.

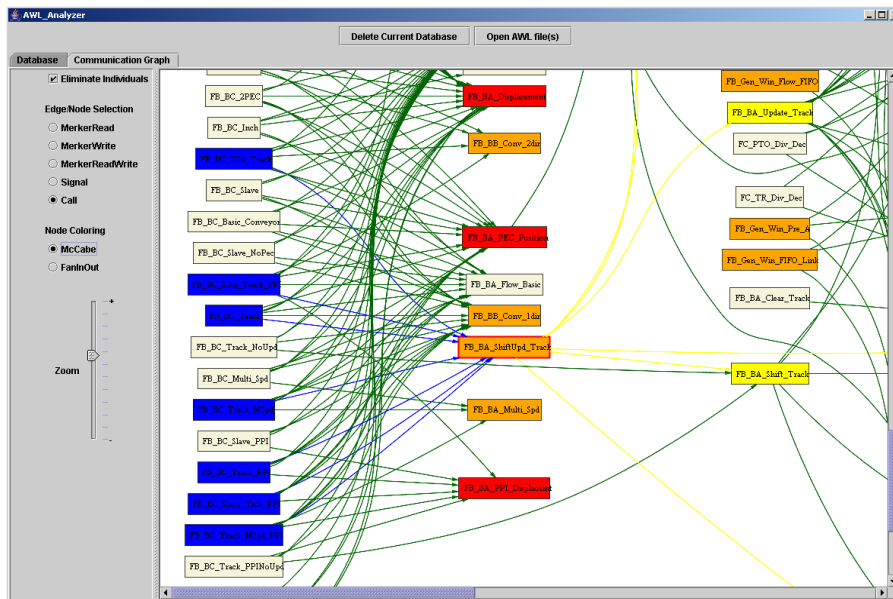


Figure 4.5: FB\_BA\_ShiftUpd\_Track clicked and direct neighbors colored

Especially with larger graphs, it can still be hard to get a good overview of all dependants and dependencies of a node. It is possible to zoom out, but it is prob-

<sup>7</sup>It is possible to set another depth (standard 1 level for head and 1 level for tail coloring, thus only coloring direct neighbors) for coloring dependants and dependencies in ColorDependencies.java of AWL Analyzer.

ably better to create a subgraph for a node and its dependants/dependencies by double clicking the node. Besides the elimination of ‘uninteresting’ nodes, it also applies the layout algorithm on the new graph again. An example of a subgraph for FB\_BA\_ShiftUpd\_Track is shown in Figure 4.6

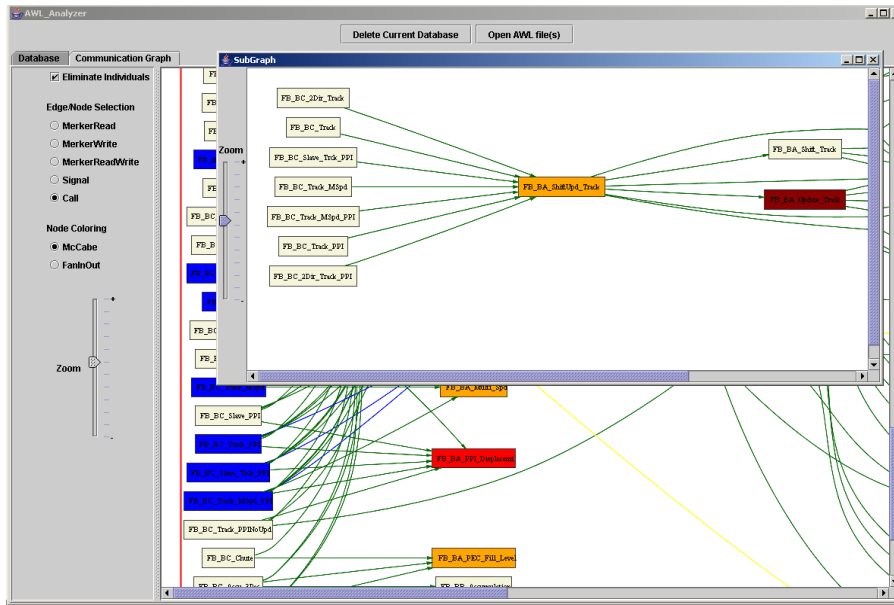


Figure 4.6: Subgraph for FB\_BA\_ShiftUpd\_Track and dependencies

#### 4.3.4 Draw all interactions with node

It can be useful to see all merker-reads and merker-writes for a given component or merker. As stated in Section 4.3, requesting the MerkerReadWrite graph can take quite some time. This is the reason that an extra option was built into AWL Analyzer to draw both merker-reads and merker-writes in the MerkerRead or MerkerWrite graph. To do so, select a component or merker in the graph by clicking it. Rightclick -> ‘draw all interactions with node’. AWL Analyzer now adds missing edges and applies the layout algorithm on the new graph.

## 5 Questionnaire

Numerous metrics and rankings can be extracted from STL components using AWL Analyzer, but what metrics are most useful to identify complex STL components? Is interaction complexity (merker operations in particular) really the most important indicator?

In order to find answers to these questions, a questionnaire was set up and sent to the STL programmers at Vanderlande. This questionnaire consisted of a number of questions that covered the main question from different viewpoints. The programmers had not seen the results of the ranking of AWL Analyzer, so objectivity was guaranteed.

The questions and their scope will be addressed in the following section. A summary of the results of the returned questionnaires will be presented afterwards. Finally, attempts are made to correlate those results with the metrics produced by the AWL Analyzer.

### 5.1 Questions

The questionnaire consisted of four questions. These covered coding experience of the programmer, rating of components based on their perceived complexity, and rating of metrics based on their perceived usefulness.

The first question was intended to identify the coding experience of a programmer:

Q1)  
Amount of years of coding experience: .....  
Amount of years of coding experience with STL (or alike): .....  
Amount of years of coding experience with STL at Vanderlande: .....

In case experienced programmers base their perception of complexity on other metrics than less experienced programmers, this can be identified.

The second question was intended to identify which components of the W&D library of Vanderlande<sup>8</sup> were perceived as complex by the STL programmers. The question consists of two parts. Question 2a requests to rate a number of components based on perceived complexity:

Using the following rating scheme from 1 to 10 for the complexity and interaction of components:

---

<sup>8</sup>37th revision, 397 components

1		
2		
3	=	<b>Easy</b>
4		Easy and fast to understand by an STL-programmer who did not work on this component before. Chance that problems (errors) occur in this or other components when updating this component is very small
5		
6	=	<b>Moderate</b>
		Average component. Not very complex or not much interaction with other components, but a small possibility that problems (errors) occur when updating this component.
7		
8	=	<b>Complex</b>
		More complex component or component with relatively high interaction. Chance and impact of errors during updates to this component are higher than average.
9		
10	=	<b>Very complex</b>
		The say 3 percent most complex and most used components.

Q2a) Can you rate these 16 components from 1 to 10 please? If you don't know the component, you can leave it blank.

- FC\_Record\_Index: .....
- FC\_Log\_Event: .....
- FC\_FIFO: .....
- FB\_OP77\_Controller: .....
- FB\_Induct\_On\_Reserve: .....
- FB\_DP\_DP\_Send\_Rcv: .....
- FB\_BC\_TR\_L\_Win: .....
- FB\_BB\_Slave\_Junction: .....
- FB\_BB\_Scanner: .....
- FB\_BB\_Divert\_PTO: .....
- FB\_BB\_Conv\_1dir: .....
- FB\_BA\_Update\_Track: .....
- FB\_BA\_ShiftUpd\_Track: .....
- FB\_BA\_PPI\_Displacemt: .....
- FB\_BA\_PEC\_Position: .....
- FB\_BA\_Divert\_Seq\_1: .....

The components requested to rate here basically all had a top-3 position on a given metric (McCabe complexity, call- and merker-interaction<sup>9</sup>, lines of code, amount of revisions, comment ratio).

Question 2b requests to give a number of components on the different suggested complexity levels:

Q2b) Can you give (at least) 2 Easy, 2 Moderate, 3 Complex and 3 Very complex components who are not in the list above? That is if you think that at least the given amount of components belongs to that category.

Easy:

Moderate:

Complex:

Very complex:

<sup>9</sup>At the moment the questionnaire was constructed, instantiation interaction was not considered yet

If a number of programmers identify a given component as very complex while it does not score high on any ('interesting') metric, this can indicate missing or incomplete metrics.

The third question was meant to find out what the STL programmers thought were the most important metrics to use in a metrics set:

There is a number of metrics that can be collected on STL code in order to say something about the complexity, importance and maintainability of a component. Some will be more useful than others. Using the following rating:

- 1 = totally not useful
- 2 = may be useful, but low importance
- 3 = probably useful and moderate importance
- 4 = useful and important
- 5 = definitely very useful and important

Q3) Can you rate the following metrics from 1 to 5 based on your perceived value of usefulness in a metrics set?

- Size of component (lines of code): .....
- Amount of decision points<sup>10</sup> in this component: .....
- Amount of calls from this component to other components: .....
- Amount of calls from other components to this component: .....
- Amount of revisions of this component: .....
- Amount of distinct programmers for this component: .....
- Rate of comment (lines of comment/lines of code): .....
- Amount of markers read by component: .....
- Amount of markers written by component: .....

The fourth question finally requested some general remarks on the matter:

Q4) If you have more ideas on metrics to collect or more comments on what determines complexity and maintainability of a component, please write them here or mail them to me on [b.seghers@student.tue.nl](mailto:b.seghers@student.tue.nl)

## 5.2 Results questionnaire

Eventually, 7 programmers filled out the questionnaire and returned it. The results are presented in the tables below:

### Results Question 1

Question	P1	P2	P3	P4	P5	P6	P7
Amount of years of coding experience	3	15	15	9	7	8	5,5
Amount of years of coding experience with STL (or alike)	3	15	15	9	2,5	8	3,5
Amount of years of coding experience with STL at Vanderlande	2,5	15	5	0	2,5	8	3

P1-P7: PLC programmers 1-7

<sup>10</sup>The amount of different 'paths' that can be taken in a piece of code. For example by if-then, if-then-else, do-while, jump-statements. The higher this amount, the more likely that it is difficult to keep track of the behavior of the piece of code.

Results Question 2a

Component	MC	F-IO	LOC	Rev	Mer-IO	CR	IIO	IC	PLC Programmers							Avg
									1	2	3	4	5	6	7	
FC_Record_Index	41	0	895	3	0	1,36	0	0	8	8	9	7	7	8	6	7,57
FC_Log_Event	4	32400	129	2	0	1,27	10	170	4	3	4	7	5	3	6	4,57
FC_FIFO	30	0	558	19	0	1,19	3	9	7	8	10	7	6	10	7	7,86
FB_OP77_Controller	51	459	1434	0	0	1,87	15	24	9	10	7	8	8	8	8	8,33
FB_Induct_On_Reserve	14	0	371	12	163296	1,39	22	340	6	3	5	7	6	4	5	5,14
FB_DP_DP_Send_Rcv	28	0	562	18	0	0,90	6	6	8	8	8	7	7	10	10	8,29
FB_BC_TR_L_Win	2	0	256	17	648	0,97	18	176	5	7						5
FB_BB_Slave_Junction	6	0	365	13	2160000	1,62	126	1350	6	6	7	6	5	4	2	5,14
FB_BB_Scanner	19	684	451	5	45619	1,15	144	476	5	6	6					5,25
FB_BB_Divert_PTO	7	0	253	9	984375	1,29	22	646	6	6						5,33
FB_BB_Conv_1dir	11	0	604	50	52468416	1,95	140	4214	8	7	8	7	5	6	2	6,14
FB_BA_Update_Track	60	10140	1330	-1	11760	1,30	104	462	9	10	9	8	8	10	5	8,43
FB_BA_ShiftUpd_Track	11	19404	337	-1	2816	2,08	0	220	6	8	9	8	7	8	4	7,14
FB_BA_PPI_Displacem	31	0	476	13	74431	1,21	7	160	8	6	6	8	7	8	4	6,71
FB_BA_PEC_Position	37	8325	742	15	1807117	1,53	20	672	9	10	9	7	6	10	9	8,33
FB_BA_Divert_Seq_1	15	0	354	7	486000	0,14	20	496	7	6						6

Component: Component name  
 MC: McCabe complexity  
 F-IO: Fan-In/Out complexity  
 LOC: Lines Of Code  
 Rev: Amount of revisions  
 Mer-IO: Merker-In/Out complexity  
 CR: Comment Ratio  
 IIO: Instantiation-In/Out complexity  
 IC: Interaction complexity  
 Avg: Average grade

Results Question 2b

Component	MC	F-IO	LOC	Rev	Mer-IO	CR	IIO	IC	PLC Programmers							Avg
									1	2	3	4	5	6	7	
FB_BA_PEC_Trigger	30	7680	547	16	17280	1,70	24	228	10	8						9
FB_OP_Controller	53	477	1261	14	0	1,40	0	12	10	10					10	10
FB_Diagnose_ASILPF	31	0	697	5	0	0,87	0	40		10						10
FB_ODP											10					10
FB_Decision_Points											10	8				9
FB_Diagnose_HW	11	0	377	1	0	1,60	0	4							10	10
FB_Copy_Records	18	288	460	3	0	1,19	0	7	8							8
FB_Diagnose_DP	14	0	294	2	0	1,38	0	12	8							8
FB_BA_Shift_Track	9	2025	201	15	20736	1,08	16	84		8						8
FB_Mailbox_ASILPF	15	135	314	5	0	0,93	0	80			8				6	7
FB_Induct_On_ResLink	18	3528	461	14	209952	1,34	108	462			8			8		8
FB_BA_SDS_Seq												8				8
FB_VertiSorter												8				8
FB_BA_Displacement	22	0	271	6	12672	1,07	24	180						8		8
FB_BA_Flow_Basic	6	0	158	6	150	2,63	6	22	6	6				6		6
FB_BA_Cascade	4	0	93	10	64	2,02	4	6	6	3	6					5
FC_Read_Track	8	0	138	4	0	2,04	0	48		6						6
FB_BC_Basic_Conveyor	2	800	227	7	0	0,99	0	117		6						6
FB_BA_PEC_Stop	13	0	214	7	51597	1,83	63	210			6					6
FB_BC_2Dir												6				6
FB_Area_StSt_Mgr												6				6
FB_VZS													6			6
FB_Host_Send/Receive													6			6
FB_System_Facilities	13	117	397	10	0	0,00	0	3							6	6
FC_ANY_to_ANY	1	0	38	3	0	0,95	0	0	3							3
FC_INT_To_4_CHARS	2	8	83	2	0	1,61	0	12	3							3
FC_4_CHARS_To_INT	2	0	84	3	0	1,31	0	8		3						3
FB_Generate_Window	4	0	112	6	0	2,12	0	3			3					3
FB_Detect_E_Stops	2	0	128	4	0	1,25	0	10								3
FC_On_Off_Dly_Timer	7	0	185	4	0	3,20	0	6					3			3
FC_Writ_ID_to_PIC_DB													3			3
FB_Cabinet_Manager														3		3
FB_BC_Conveyor														3		3
FB_BC_Track	2	288	266	10	0	0,83	0	126							3	3
FB_Delay_On-Off	6	0	73	3	0	1,59	0	0							3	3

Component: Component name  
 MC: McCabe complexity  
 F-IO: Fan-In/Out complexity  
 LOC: Lines Of Code  
 Rev: Amount of revisions  
 Mer-IO: Merker-In/Out complexity  
 CR: Comment Ratio  
 IIO: Instantiation-In/Out complexity  
 IC: Interaction complexity  
 1-7: PLC programmers 1-7  
 Avg: Average grade

### Results Question 3

Metric	P1	P2	P3	P4	P5	P6	P7	Avg
Size of component (lines of code):	3	3	2	2	2	2	2	2,29
Amount of decision points in this component:	4	5	4	4	4	5	5	4,43
Amount of calls from this component to other components:	5	5	3	3	3	3	4	3,71
Amount of calls from other components to this component:	5	4	2	2	3	3	2	3
Amount of revisions of this component:	3	4	4	4	4	4	4	3,86
Amount of distinct programmers for this component:	2	4	3	3	4	3	3	3,14
Rate of comment (lines of comment/lines of code):	2	5	3	5	5	4	2	3,71
Amount of markers read by component:	5	4	3	4		3	2	3,5
Amount of markers written by component:	5	4	3	4		3	2	3,5

Avg: Average  
P1-P7: PLC programmers 1-7

### Results Question 4

In component FB\_BA\_Update\_Track:

Network KA: Search for data in open window offset + Remove Double Data is very complex

Network LA: Determine update data is very complex

Variables with t\_Data, s\_Data, s\_Update\_Data are passed from network to network and at a given moment you are lost (but not in tracking).

Block changed to often which makes it loose its grip.

In component FB\_BA\_PEC\_Position:

shift function is very complex

hand over offset is very complex

Block changed to often which makes it loose its grip.

Maybe useful to check the number of used pointers. In general, the use of pointers is difficult for a lot of programmers.

Differences in perceived complexity of a given component exist amongst programmers (Question 2). On average however, most perceived complexities do not differ more than one grade from the average. This indicates that most programmers find the same (type of) components complex or easy. The same holds for the importance of metrics in question 3. This led to the decision to define perceived complexity for a component as the average of the perceived complexities of the programmers.

$$\text{Perceived Complexity Component}_i = \text{Average}(\text{Perceived complexities programmers for component}_i)$$

And perceived importance of a metric as the average of the perceived importance by the programmers.

$$\text{Perceived Importance Metric}_j = \text{Average}(\text{Perceived importance programmers for metric}_j)$$

Perceived complexity and perceived importance will be used in attempts to correlate the results from the questionnaire with the metrics produced by AWL Analyzer. These attempts are addressed in the following section.

### 5.3 Correlation results AWL-Analyzer and questionnaire

The questionnaire was set up to identify what metrics would be most useful in identifying complex STL components. Actually, to find out if metrics constructed by AWL Analyzer could be used to identify potentially complex and error-prone components. If this was the case, then those (combinations of) metric-values for the components should correlate with the perceived complexities for the components (reported by the STL programmers).

Initially, the thought was that interaction complexity (merker interaction in particular) would probably be a good measure for identifying complex STL components. As will be explained in this section however, this seems not the case. McCabe complexity on the other hand works very well.

The correlation test used (hypothesis test simple linear regression) will be explained first. The correlation of numerous metric-values for components with the perceived complexity by the STL programmers for the components will be addressed in the section afterwards. Finally, the conclusions drawn (as mentioned above) will be explained.

#### 5.3.1 Hypothesis test simple linear regression

Regression analysis is a statistical technique for modeling and investigating the relationship between two or more variables. In this case, an attempt is made to investigate the relationship between two variables. Perceived complexity on the one hand and a metric (produced by AWL Analyzer) on the other hand. If these two variables are used as x- and y-coordinates for the points in an xy-scatter plot, then how are these points scattered around a straight line?

Mathematically, this means ‘how do these points fit in the so-called simple linear regression model?’:

$$Y = \beta_0 + \beta_1 x + \epsilon \tag{1}$$

Y is the y-coordinate of the (straight) line for a given x,  $\beta_0$  and  $\beta_1$  are the so-called regression coefficients, and  $\epsilon$  is the random error term.

The best estimates for the regression coefficients ( $\beta_0$  and  $\beta_1$ ) can be computed with the method of least squares. With this method, the sum of the squares of the vertical deviations (points-line) is minimal. For details on this method, see [MR99].

In order to test how well the points fit in the simple linear regression model, statistical hypotheses need to be tested about the model parameters. Furthermore, confidence intervals should be constructed. The coefficients and parameters used and presented in the following sections will be addressed briefly. For details on the computation, see [MR99].

Statistically significant relationship at the xx% confidence level:	Test if a straight line model is adequate to relate var1 and var2 with xx% confidence. What is the chance that the observations were not pure ‘luck’?
Standard error of estimates:	The standard deviation $\sigma$ of the sample mean.

The parameters and hypotheses tests presented in the following sections can all be calculated by StatGraphics [StatGr] for a given data set. This package was therefore used for the tests in the following sections.

### 5.3.2 Correlation

Metrics can be collected for all components in the W&D library of Vanderlande (with AWL Analyzer), and perceived complexity is available for a number of components. The perceived complexity of components ranked in question 2a of the questionnaire are more accurate than those in question 2b however. Simply because they are ranked by all (or at least most of the) STL programmers. Therefore it was useful to try to correlate the components from question 2a and question 2b separately. These correlation attempts lead to basically the same results for question 2a and 2b (although the standard error of the estimates is lower for question 2a than for question 2b). This is why only the results of components in question 2a are presented here.

The metrics used to try to correlate with perceived complexity are McCabe complexity, lines of code, amount of revisions, comment ratio, merker-IO, fan-IO, instantiation-IO and interaction complexity. McCabe and lines of code seem to have a strong correlation with perceived complexity and will be addressed first. The other metrics do not seem to have a strong correlation and will be addressed in the two sections afterwards.

#### Perceived Complexity – McCabe Complexity and LOC

Figures 5.1 and 5.2 show the XY scatter plots of McCabe complexity and lines of code against perceived complexity respectively. The 16 dots represent the 16 components from question 2a. As can be seen, a correlation seems to exist between McCabe complexity and perceived complexity on the one hand, and lines of code and perceived complexity on the other hand.

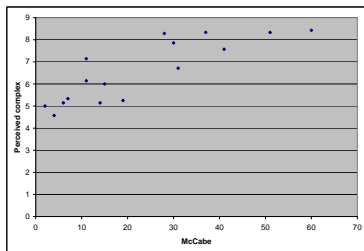


Figure 5.1: XY Scatter of Perceived- and McCabe Complexity

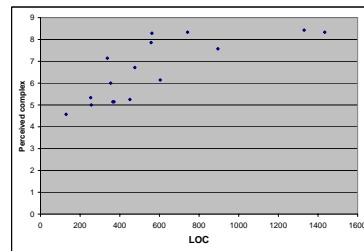


Figure 5.2: XY Scatter of Perceived Complexity and Lines of code

In order to check these hypotheses, both data tables (McCabe–Perceived Complexity and Lines Of Code–Perceived Complexity) were fitted in StatGraphics [StatGr]. The results of these fits can be found in Appendices A.1 and A.2. The result in Appendix A.1 indicates that there is a statistically significant relationship between perceived complexity and McCabe complexity (Perceived Complexity = 5,02737 + 0,0670596\*McCabe Complexity) at the 99% confidence level. The result in Appendix A.2 indicates that there is a statistically significant relationship between perceived complexity and lines of code (4,91211 +

0,00288925\*Lines Of Code) at the 99% confidence level. The Standard Error of Estimates (0,775679 versus 0,947149) is lower for McCabe-perceived than for LOC-perceived however. This indicates that McCabe complexity is a better indicator for perceived complexity than lines of code. Thus:

$\text{Perceived Complexity} = 5,02737 + 0,0670596 * \text{McCabe Complexity}$
$\text{Standard Error of Estimates} = 0,775679$

This means that components with McCabe complexities of 14, 44, and 74 will on average be identified as moderate, complex, and very complex respectively<sup>11</sup>.

### Perceived Complexity – Amount of revisions and Comment Ratio

Figures 5.3 and 5.4 show the XY scatter plots of amount of revisions and comment ratio against perceived complexity respectively. Opposed to the plots of McCabe and LOC, there does not seem to exist a correlation between amount of revisions and perceived complexity nor comment ratio and perceived complexity.

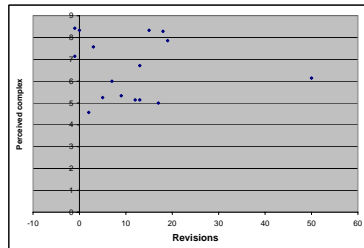


Figure 5.3: XY Scatter of Perceived Complexity and Revisions

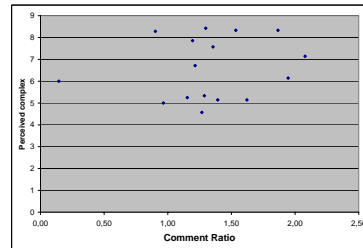


Figure 5.4: XY Scatter of Perceived Complexity and Comment ratio

The results from fitting both data sets in StatGraphics [StatGr] (Appendices A.3 and A.4) indeed indicate that there is no statistically significant relationship at the 90% confidence level in both data sets.

### Perceived Complexity – Merker-IO, Fan-IO, Instantiation-IO and Interaction Complexity

The interaction metrics were expected to be good indicators for perceived complexity. The XY scatter plots of merkerIO, fanIO, instantiationIO and interaction complexity against perceived complexity (Figures 5.5, 5.6, 5.7 and 5.8) did not seem to support this hypothesis however.

<sup>11</sup>Based on the perceived complexity rating used in the questionnaire: 6=moderate, 8=complex, 10=very complex

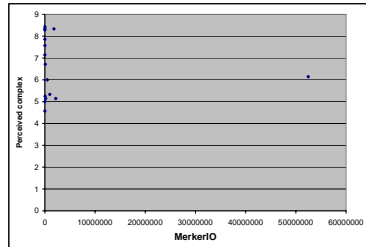


Figure 5.5: XY Scatter of Perceived- and Merker Complexity

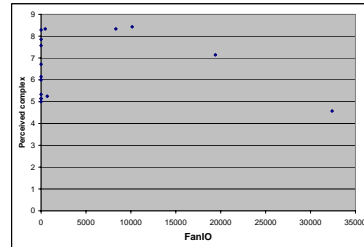


Figure 5.6: XY Scatter of Perceived- and Fan Complexity

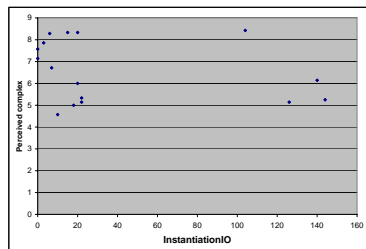


Figure 5.7: XY Scatter of Perceived- and Instantiation Complexity

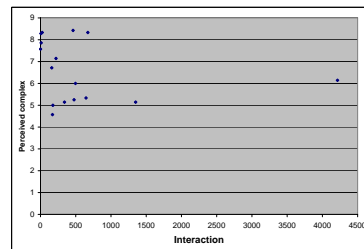


Figure 5.8: XY Scatter of Perceived- and Interaction Complexity

The results from fitting all four data sets in StatGraphics [StatGr] (Appendices A.5, A.6, A.7 and A.8) indeed all indicate that there is no statistically significant relationship at the 90% confidence level in the data sets.

### 5.3.3 Conclusions correlation results

Lines of code, but particularly McCabe complexity gave a pretty good estimation of the perceived complexity of a component. Interaction complexity (or any other interaction complexity measure) on the other hand however did not show a high correlation with perceived complexity. This was not what we expected. Interaction complexity was thought to be much more important in indicating a complex component. This could mean two things:

- The programmers had addressed the internal complexity of the components only and had not really incorporated the interaction of the components.

- Maybe, interaction was not as important as thought.

After presenting the results to the STL programmers, it became clear that they had indeed basically all addressed the internal complexity of the components. It was just extremely difficult if not impossible to also address the interaction of components in this grading. They found it very difficult to mix ‘internal’ and ‘external’ complexity into one figure. According to the programmers, both internal and external complexity could help in defining components with a high potential for refactoring however.

<p><b>McCabe complexity</b> and <b>Interaction complexity</b> can help in defining components with a high potential for refactoring.</p>
--

Based on this, the decision was made to construct the metrics set based on two main metrics. The one would be McCabe complexity for the internal complexity. The other one would be Interaction complexity for the external complexity.

## 6 Trends complexity in code

The goal of this (and follow-up) project(s) is to be able to identify STL components with a need for preventive refactoring. This can be done by retrieving metrics from those components and defining thresholds for the metrics. The AWL Analyzer can be used to collect the metrics for a given revision (usually the last one). It would however be useful to see the complexity trends of components and not only the current complexity levels. Some components just are complex (and stay complex after refactoring). If those components maintain a certain complexity level, they are usually manageable. If on the other hand, the complexity of a component is increasing continuously over time (and reaches a certain complexity level), this component might need some attention.

This section analyzes the McCabe complexity and interaction complexity trends for 19 components from the W&D library of Vanderlande. The 10 components with (currently<sup>12</sup>) the highest McCabe complexity and the 10 components<sup>13</sup> with the highest interaction complexity. Those trends indeed show that some components have steady complexity levels and others have increasing complexity levels over time. Furthermore, it became clear that AWL Analyzer could provide a visual insight in interaction structure changes of a component over time.

### 6.1 McCabe and interaction trend graphs 19 components

For 19 components, all revisions were extracted from the repository and fed to AWL Analyzer (component per component). Both McCabe complexity and interaction complexity were plotted against time for all those components.

Figure 6.1 shows the McCabe complexity over time for the 10 components with the highest (current) McCabe complexities (between 28 and 60).

---

<sup>12</sup>Release 3.7 of the W&D library

<sup>13</sup>FB\_BA\_PEC\_Position occurs in both top10 lists (McCabe and interaction)

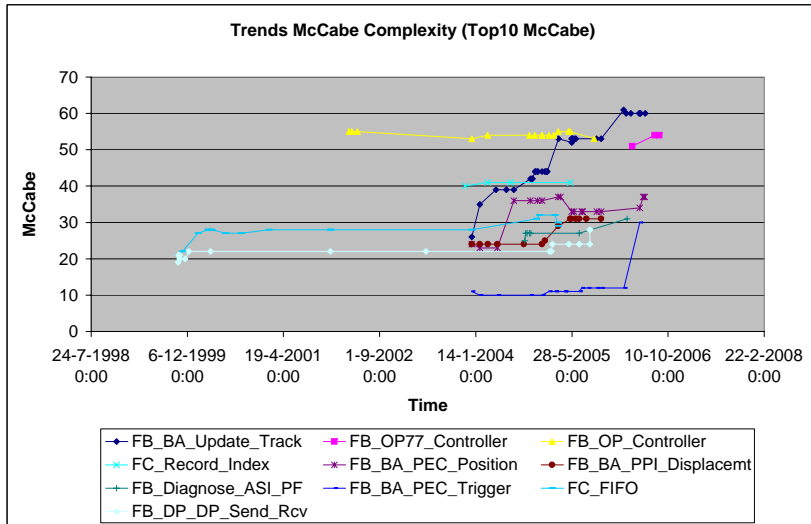


Figure 6.1: McCabe complexity trends of 10 components with highest McCabe complexity in release 3.7 of the W&D library at Vanderlande

It can be seen that components like FB\_OP\_Controller, FC\_Record\_Index and FC\_FIFO have high McCabe complexities, but seem stable. Most other components however show increases in McCabe complexity over time. This could be alarming. Especially since some of these components also show (extreme) increases in interaction complexity lately, as can be seen in Figure 6.2.

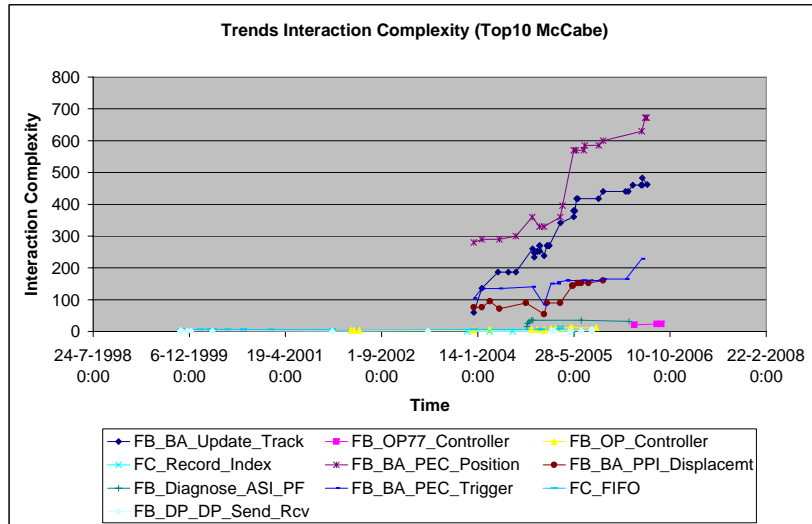


Figure 6.2: Interaction complexity trends of 10 components with highest McCabe complexity in release 3.7 of the W&D library at Vanderlande

Also the 10 components with the highest (current) interaction complexities (between 592 and 4214) show trends upwards. Figure 6.3 shows the McCabe complexity over time for those 10 components.

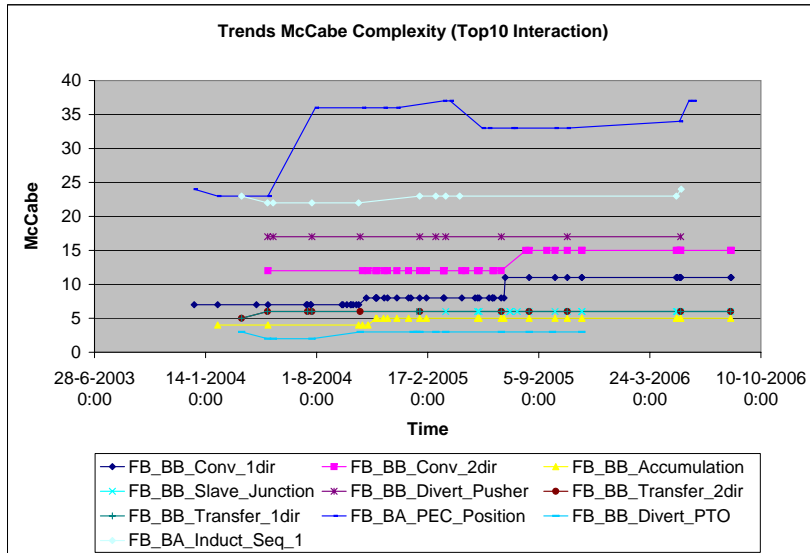


Figure 6.3: Interaction complexity trends of 10 components with highest interaction complexity in release 3.7 of the W&D library at Vanderlande

All the McCabe complexities of the 10 components with the highest (current) interaction complexities increase over time. Most alarming for these components however are their interaction complexities as can be seen in Figure 6.4.

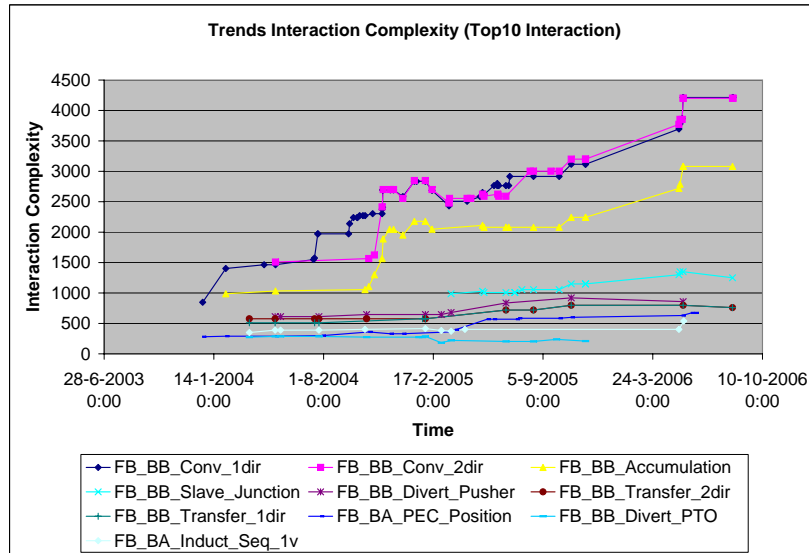


Figure 6.4: Interaction complexity trends of 10 components with highest interaction complexity in release 3.7 of the W&D library at Vanderlande

Especially FB\_BB\_Conv\_2dir, FB\_BB\_Conv\_1dir, and FB\_BA\_Update\_Track have rapidly increasing interaction complexities. Even apart from having the highest current interaction complexities. Components like these probably are or will soon become high risk components and are potential candidates for refactoring.

Another remark to be made based on Figure 6.4 is that FB\_BB\_Conv\_2dir, FB\_BB\_Conv\_1dir, and FB\_BA\_Update\_Track seem highly correlated. This might be useful for refactoring. Refactorings of these three components might be the same or alike which can save time and money. By only loading FB\_BB\_Conv\_2dir, FB\_BB\_Conv\_1dir, and FB\_BA\_Update\_Track into AWL Analyzer, it was very easy to see that their interaction structures indeed were alike. This means they call the same components, and use the same markers and instantiations for a very large part.

## 6.2 Use AWL Analyzer to visualize structure changes in component

An interesting and non accounted for side-effect of the trend analysis was the insight it gave in structure changes in components analyzed. After importing all revisions of a given component into AWL Analyzer, the graph tab visualized all interaction for the different revisions. Changes in interaction structure between different revisions could therefore be identified very easily. An example of such

a change (which led to an increase in interaction complexity) can be seen in Figures 6.5 and 6.6.

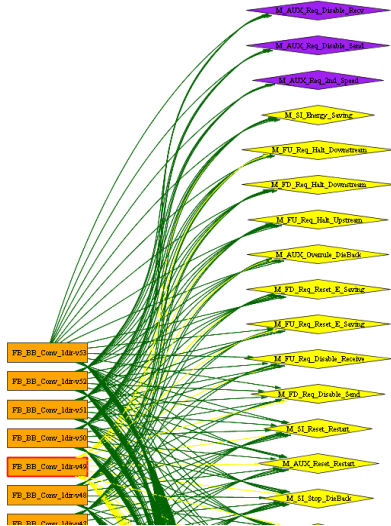


Figure 6.5: Merker writes FB\_BB.Conv\_1dir version 49

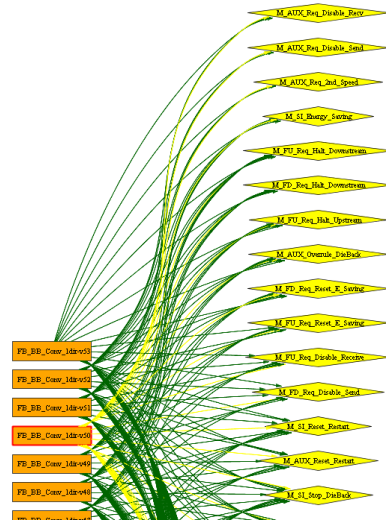


Figure 6.6: Merker writes FB\_BB.Conv\_1dir version 50

This example shows (part of) the difference in merker writes of this component between revision 49 and 50. Version 50 reads 3 merkers more than version 49 (M\_AUX\_Req\_Disable\_Recv, M\_AUX\_Req\_Disable\_Send and M\_AUX\_Req\_2nd\_Speed). Not only is it possible to see an increase in interaction complexity through revisions in a component. Also the sources of these increases can be identified rapidly and easily.

### 6.3 Conclusions

The complexity trends indeed show that some components have steady complexity levels while others have increasing complexity levels over time. This makes identifying error-prone components (with the highest priorities for refactoring) more accurate.

Furthermore, AWL Analyzer can provide a visual insight in interaction structure changes of a component over time. This makes identifying the sources of (extreme) increases in interaction complexity a lot easier.

## 7 Conclusions

The main objective for this assignment is to define metrics that signal the need for preventive refactoring. This objective was split up in three major sub-tasks and objectives.

The first task consisted of studying earlier research in the field of maintainability of PLC code and related metrics. And, if research was conducted, can this be used to improve the maintainability of PLC code at Vanderlande?

Based on literature research in section 2.2, it can be concluded that there is no indication that anything of value has been done in the field of complexity-metrics on PLC-code (STL in particular) yet.

The second task was to study the architecture of the PLC software platform, to learn what determines its maintainability, and define metrics based on these insights.

The complexity of a software component determines its maintainability for a large part. Section 2.3 addressed a number of metrics to measure the complexity of a component: file size/LOC, amount of comment, numerous metrics on revisions, McCabe complexity, and information flow metrics. Section 5.3.3 showed that McCabe complexity and information flow metrics are the most important complexity indicators for STL. McCabe complexity for the 'internal complexity', and information flow metrics for the 'interaction complexity'. Sections 4.2.1 and 4.2.2 presented details on how to calculate or retrieve these metrics from STL:

**Pseudo code McCabe complexity in STL:**

```
Decide_Count = 0;
for line 1 to last line of component
  if the line contains an operator (JumpOp/LoopOp) then
    Decide_Count = Decide_Count + 1;
  fi
rof
McCabe Complexity = Decide_Count + 1;
```

and

$$\text{Interaction Complexity} = (\text{fan-in} + \text{merker-reads} + \text{instance-reads}) * (\text{fan-out} + \text{merker-writes} + \text{instance-writes})$$

Thresholds for McCabe complexity can be calculated with the formula presented in Section 5. This formula is based on the results of a questionnaire held under a number of PLC programmers:

$$\text{Perceived Complexity} = 5,02737 + 0,0670596 * \text{McCabe Complexity}$$

$$\text{Standard Error of Estimates} = 0,775679$$

This means that components with McCabe complexities of 14, 44, and 74 will on average be identified as moderate, complex, and very complex respectively<sup>14</sup>.

Besides ‘current’ complexity levels, complexity trends were analyzed in order to make the identification of error-prone components (with the highest priorities for refactoring) more accurate. These trends show that some components have steady complexity levels (McCabe, interaction) while others have increasing complexity levels over time.

The third and last task was to specify tools to collect the metrics found in the second task/objective automatically. These tools were to be implemented in later student assignments.

Section 3.7 showed that no specific tools were available to collect metrics for STL. This is the reason the choice was made to build a simple parser from scratch in this assignment. This led to the construction of AWL Analyzer v 1.0.

AWL Analyzer can be used to collect numerous metrics (including McCabe and interaction complexity) on STL components. AWL Analyzer has sorting and selection functionality, and can visualize the interaction structures. Useful for a better insight in the code structure and to define components with the highest potential for refactoring.

Furthermore, AWL analyzer can provide a visual insight in interaction structure changes of a component over time. This makes identifying the sources of (extreme) increases in interaction complexity a lot easier.

The completion of the three subtasks led to the fulfillment of the main objective for this assignment: “To define metrics that signal the need for preventive refactoring”. Furthermore, a tool was produced that can actually collect and visualize these metrics.

---

<sup>14</sup>Based on the perceived complexity rating used in the questionnaire: 6=moderate, 8=complex, 10=very complex

## 8 Recommendations

In this section, recommendations are given for the integration of AWL Analyzer at Vanderlande, as well as further research and improvements that can be done based on the research conducted during this assignment.

### Usage and integration of AWL Analyzer

Analyzing the complexities of components is a new and extra task for Vanderlande, and will not always pay back right away. For a successful integration in the processes of Vanderlande, this task should be easy to perform, not much (extra) time should be required, and no other processes should be disturbed. Besides that, it is advisable to check the complexities of components at given moments (to make the analysis part of the protocol). This way, the analysis will not be skipped.

It would therefore be useful to **analyze a whole library using AWL Analyzer every time a new version is released of that library**. In this analysis, changes in the top10 for McCabe complexity and interaction complexity should definitely be addressed. Do complexities increase compared to the last release? Do new components pass a certain threshold?

It might also be advisable to perform a quick analysis of the other metrics and of the interaction structures to see if large changes occur compared to the previous release.

### Further research and improvements

This assignment led to a number of new questions and ideas. These will be addressed shortly in this paragraph.

**Perform research on thresholds interaction complexity:** more research on the importance of interaction complexity is required as well as how it is perceived by the PLC programmers in order to define these thresholds.

**Analyze complexity trends in more detail and automate process of retrieving:** only a short amount of time was available to analyze complexity trends and it still required a certain amount of handwork. Complexity trends seem very useful however and give a more accurate view on what components are inerrant complex and what components are becoming complex. It could therefore be an idea to automate the process of retrieving all revisions for a component from the repository and loading them into AWL Analyzer.

**Address complexity distribution within components:** does a high complexity metric come from a very small and specific part of the code? In other words, is the complexity concentrated? If so, it might be relatively easy to lower the overall complexity of the component.

**Address complexity migration within components:** this research combines the complexity distribution with trends. Does the complexity migrate through the code in different revisions, or does it come from a specific part all the time?

**Resolve performance problems:** as addressed in Section 4.3.2, there are some performance problems when it comes to the layout algorithm. Research

could be done in this area. Also the interaction with the database can probably be optimized or another database engine could be used to speed up things.

AWL Analyzer can be used for most of the ideas mentioned in this paragraph. This is the reason AWL Analyzer was transferred to Vanderlande Industries including all necessary documentation for further development.

## 9 Evaluation

In this section, this assignment will be evaluated. First, the slight assignment change will be addressed. Then, something will be written about the usefulness of the rapid prototyping process used. Finally, the usability of the results from this assignment will be addressed.

### 9.1 Change in assignment

The main objective for this assignment is to define metrics that signal the need for preventive refactoring. During the rapid prototyping process, it became clear that a good insight in the (current) interaction structure of the PLC platform maybe was even more important however. Most feedback and requests on prototypes of AWL Analyzer were directed at more and better insights in this structure. This is the reason interaction metrics and interaction structure are addressed in quite some detail in this paper. Luckily it was possible to combine both objectives in such a way that they could both be met.

### 9.2 Rapid prototyping process

An iterative process called rapid prototyping was used in this assignment. It consists of building a prototype, show it to your client, get feedback, and build a new prototype based on this feedback. This process was used to show Vanderlande what possibilities there were for the collection of metrics on STL code. They could then give indications about what type of metrics and information were most useful for them.

Not only did this process lead to AWL Analyzer, it also revealed that a good insight in the interaction structure of the PLC platform was even more important than retrieving metrics for preventive refactoring.

Rapid prototyping led to a better involvement of all parties since the metrics and the information became more tangible. Not only were we talking about a certain type of metric, we could extract actual values for this metric from STL-components of Vanderlande.

### 9.3 Usability results

The developers at Vanderlande are very enthusiastic about the information they can collect on their STL libraries now. AWL Analyzer can become a very useful tool for analyzing the PLC platform on a regular basis.

Furthermore, the results on how to extract metrics from STL code are totally new and can become very useful. Definitely the McCabe complexity for an STL component, since there seems to be a high correlation between this complexity and the actual complexity perceived by the programmers.

## 10 Bibliography

### References

- [CMU06] Carnegie Mellon Software Engineering Institute “Software Technology Roadmap - Cyclomatic Complexity”, [http://www.sei.cmu.edu/str/descriptions/cyclomatic\\_body.html](http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html), contacted Aug-14-2006
- [Dav05] Ryan Davis, Seattle.rb, *Polishing Ruby presentation*, Seattle, 2005
- [HK84] Henry S. and Kafura, D., *The evaluation of software Systems’ Structure Using Quantitative Software Metrics*, Software Practice and Experience, 14, (6), pp. 561–573, 1984
- [IEEE88] IEEE 982.2, *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*, A25. Data of Information Flow Complexity. p. 112., 1988
- [IEEE93] IEEE std 1219, *IEEE Standard for software maintenance*, Los Alamitos, CA. 1993
- [Jon94] Capers Jones, *Software metrics good, bad and missing*, Software Productivity Research Inc., New England Executive Park, Burlington, MA, Sept. 1994
- [Kit88] Barbera A. Kitchenham, *An evaluation of software structure metrics*, City for Software Reliability, in Proceedings of IEEE 12th Annual International Computer Software and Applications Conference COMPSAC ’88, Chicago Ill., pp. 36976., October 1988
- [MAL05] Jorge Cláudio Cordeiro Pires Mascena, Eduardo Santana de Almeida, Sílvia Romero de Lemos Meira, *A comparative study on software reuse metrics and economic models from a traceability perspective*, Federal University of Pernambuco, C.E.S.A.R Recife Center for Advanced Studies and Systems, IEEE Information Reuse and Integration, pp. 72–77, Las Vegas, USA, 15-17 Aug. 2005
- [McC76] T.J. McCabe, *A complexity measure*, IEEE Trans. Software Eng., vol. SE-2, No. 4, pp. 308-320, Dec. 1976
- [MR99] D.C. Montgomery, and G.C. Runger, *Applied Statistics and Probability for Engineers*, 2nd. ed., Wiley 1999
- [RH68] Rubey, R.J.; Hartwick, R.D., *Quantitative Measurement Program Quality*, ACM, National Computer Conference pp., 671-677, 1968
- [RSM06] Resource Standard Metrics, Metrics Definitions,

[http://msquaredtechnologies.com/m2rsm/docs/rsm\\_metrics\\_narration.htm](http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm),  
Contacted Aug-16-2006

[Sie98] Siemens AG, *SIMATIC S7 Statement List (STL) for S7-300 and S7-400 programming: reference manual*, Release 01, Germany, 1998

[SST05] S7 Standardization Team, *PLC Architectural Design - S7 Product Package*, Vanderlande Industries Nederland B.V., Research & Development, Document number: 0G0026.453.00003.EN.012, 23-05-2005

[StatGr] Statistical Graphics Corp., *STATGRAPHICS Plus v5.1*, 1994-2001

[Wey88] Elaine J. Weyuker, *Evaluating Software Complexity Measures*, IEEE Trans. Software Eng., vol. 14, No. 9, pp. 1357-1365, Sep. 1988

[WikL06] Wikipedia, the free encyclopedia,  
[http://en.wikipedia.org/wiki/Source\\_lines\\_of\\_code](http://en.wikipedia.org/wiki/Source_lines_of_code), Contacted Aug-16-2006

[WikP06] Wikipedia, the free encyclopedia,  
[http://en.wikipedia.org/wiki/Programmable\\_logic\\_controller](http://en.wikipedia.org/wiki/Programmable_logic_controller), Contacted  
Jul-31-2006

[Zuse06] Horst Zuse "History of Software Measurement", <http://irb.cs.tu-berlin.de/zuse/metrics/3-hist.html>, contacted Aug-16-2006

## A Regression analysis on perceived complexity

### A.1 Perceived complexity - McCabe

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived Complexity  
Independent variable: McCabe  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	5,02737	0,323829	15,5248	0,0000
Slope	0,0670596	0,011215	5,97947	0,0000

#### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	21,5124	1	21,5124	35,75	0,0000
Residual	8,42349	14	0,601678		
Total (Corr.)	29,9359	15			

Correlation Coefficient = 0,847712  
R-squared = 71,8616 percent  
R-squared (adjusted for d.f.) = 69,8517 percent  
Standard Error of Est. = 0,775679  
Mean absolute error = 0,597384  
Durbin-Watson statistic = 2,78479 (P=0,0529)  
Lag 1 residual autocorrelation = -0,394964

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived Complexity and McCabe. The equation of the fitted model is

$$\text{Perceived\_Complexity} = 5,02737 + 0,0670596 \cdot \text{McCabe}$$

Since the P-value in the ANOVA table is less than 0.01, there is a statistically significant relationship between Perceived Complexity and McCabe at the 99% confidence level.

The R-Squared statistic indicates that the model as fitted explains 71,8616% of the variability in Perceived\_Complexity. The correlation coefficient equals 0,847712, indicating a moderately strong relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 0,775679. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 0,597384 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.

## A.2 Perceived complexity - Lines Of Code

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived Complexity

Independent variable: LOC  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	4,91211	0,446499	11,0014	0,0000
Slope	0,00288925	0,000656477	4,40114	0,0006

### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	17,3767	1	17,3767	19,37	0,0006
Residual	12,5593	14	0,897091		
Total (Corr.)	29,9359	15			

Correlation Coefficient = 0,76188

R-squared = 58,0462 percent

R-squared (adjusted for d.f.) = 55,0495 percent

Standard Error of Est. = 0,947149

Mean absolute error = 0,7571

Durbin-Watson statistic = 2,21659 (P=0,3484)

Lag 1 residual autocorrelation = -0,108515

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived Complexity and LOC. The equation of the fitted model is

$$\text{Perceived Complexity} = 4,91211 + 0,00288925 \cdot \text{LOC}$$

Since the P-value in the ANOVA table is less than 0.01, there is a statistically significant relationship between Perceived Complexity and LOC at the 99% confidence level.

The R-Squared statistic indicates that the model as fitted explains 58,0462% of the variability in Perceived Complexity. The correlation coefficient equals 0,76188, indicating a moderately strong relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 0,947149. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 0,7571 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.

### A.3 Perceived complexity - Revisions

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived Complexity  
Independent variable: Revisions  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	6,71966	0,506764	13,2599	0,0000
Slope	-0,0116129	0,0289727	-0,400821	0,6946

#### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	0,339633	1	0,339633	0,16	0,6946
Residual	29,5963	14	2,11402		
Total (Corr.)	29,9359	15			

Correlation Coefficient = -0,106514  
R-squared = 1,13453 percent  
R-squared (adjusted for d.f.) = -5,92729 percent  
Standard Error of Est. = 1,45397  
Mean absolute error = 1,22012  
Durbin-Watson statistic = 2,24968 (P=0,3257)  
Lag 1 residual autocorrelation = -0,144512

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived\_ Complexity and Revisions. The equation of the fitted model is

$$\text{Perceived\_Complexity} = 6,71966 - 0,0116129 \cdot \text{Revisions}$$

Since the P-value in the ANOVA table is greater or equal to 0.10, there is not a statistically significant relationship between Perceived\_Complexity and Revisions at the 90% or higher confidence level.

The R-Squared statistic indicates that the model as fitted explains 1,13453% of the variability in Perceived Complexity. The correlation coefficient equals -0,106514, indicating a relatively weak relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 1,45397. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 1,22012 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.

## A.4 Perceived complexity - Comment ratio

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived Complexity

Independent variable: Comment Ratio  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	5,88355	1,13216	5,19676	0,0001
Slope	0,523715	0,80929	0,647129	0,5280

### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	0,869453	1	0,869453	0,42	0,5280
Residual	29,0665	14	2,07618		
Total (Corr.)	29,9359	15			

Correlation Coefficient = 0,170422

R-squared = 2,90438 percent

R-squared (adjusted for d.f.) = -4,03102 percent

Standard Error of Est. = 1,44089

Mean absolute error = 1,20626

Durbin-Watson statistic = 2,2662 (P=0,2819)

Lag 1 residual autocorrelation = -0,149505

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived Complexity and Comment Ratio. The equation of the fitted model is

$$\text{Perceived Complexity} = 5,88355 + 0,523715 \cdot \text{Comment Ratio}$$

Since the P-value in the ANOVA table is greater or equal to 0.10, there is not a statistically significant relationship between Perceived Complexity and Comment Ratio at the 90% or higher confidence level.

The R-Squared statistic indicates that the model as fitted explains 2,90438% of the variability in Perceived Complexity. The correlation coefficient equals 0,170422, indicating a relatively weak relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 1,44089. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 1,20626 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.

## A.5 Perceived complexity - MerkerIO

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived Complexity  
Independent variable: MerkerIO  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	6,699	0,39515	16,9531	0,0000
Slope	-0,000497181	0,000680889	-0,730193	0,4773

### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	1,09827	1	1,09827	0,53	0,4773
Residual	28,8377	14	2,05983		
Total (Corr.)	29,9359	15			

Correlation Coefficient = -0,191539  
R-squared = 3,66872 percent  
R-squared (adjusted for d.f.) = -3,21208 percent  
Standard Error of Est. = 1,43521  
Mean absolute error = 1,22118  
Durbin-Watson statistic = 2,25157 (P=0,3007)  
Lag 1 residual autocorrelation = -0,144969

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived Complexity and MerkerIO. The equation of the fitted model is

$$\text{Perceived Complexity} = 6,699 - 0,000497181 \cdot \text{MerkerIO}$$

Since the P-value in the ANOVA table is greater or equal to 0.10, there is not a statistically significant relationship between Perceived Complexity and MerkerIO at the 90% or higher confidence level.

The R-Squared statistic indicates that the model as fitted explains 3,66872% of the variability in Perceived Complexity. The correlation coefficient equals -0,191539, indicating a relatively weak relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 1,43521. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 1,22118 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.

## A.6 Perceived complexity - FanIO

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived Complexity

Independent variable: FanIO  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	6,71604	0,390701	17,1897	0,0000
Slope	-0,013621	0,0158369	-0,860079	0,4042

### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	1,50238	1	1,50238	0,74	0,4042
Residual	28,4336	14	2,03097		
Total (Corr.)	29,9359	15			

Correlation Coefficient = -0,224023

R-squared = 5,01865 percent

R-squared (adjusted for d.f.) = -1,76573 percent

Standard Error of Est. = 1,42512

Mean absolute error = 1,235

Durbin-Watson statistic = 2,00577 (P=0,4877)

Lag 1 residual autocorrelation = -0,0247668

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived Complexity and FanIO. The equation of the fitted model is

$$\text{Perceived Complexity} = 6,71604 - 0,013621 \cdot \text{FanIO}$$

Since the P-value in the ANOVA table is greater or equal to 0.10, there is not a statistically significant relationship between Perceived Complexity and FanIO at the 90% or higher confidence level.

The R-Squared statistic indicates that the model as fitted explains 5,01865% of the variability in Perceived Complexity. The correlation coefficient equals -0,224023, indicating a relatively weak relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 1,42512. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 1,235 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.

## A.7 Perceived complexity - Instantiations

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived  
Independent variable: Instantiations  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	6,85127	0,452347	15,1461	0,0000
Slope	-0,00665196	0,00686147	-0,969466	0,3488

### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	1,88326	1	1,88326	0,94	0,3488
Residual	28,0527	14	2,00376		
Total (Corr.)	29,9359	15			

Correlation Coefficient = -0,250818  
R-squared = 6,29098 percent  
R-squared (adjusted for d.f.) = -0,402523 percent  
Standard Error of Est. = 1,41554  
Mean absolute error = 1,1501  
Durbin-Watson statistic = 2,46861 (P=0,1371)  
Lag 1 residual autocorrelation = -0,252743

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived and Instantiations. The equation of the fitted model is

$$\text{Perceived} = 6,85127 - 0,00665196 \cdot \text{Instantiations}$$

Since the P-value in the ANOVA table is greater or equal to 0.10, there is not a statistically significant relationship between Perceived and Instantiations at the 90% or higher confidence level.

The R-Squared statistic indicates that the model as fitted explains 6,29098% of the variability in Perceived. The correlation coefficient equals -0,250818, indicating a relatively weak relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 1,41554. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 1,1501 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.

## A.8 Perceived complexity - Interaction

Regression Analysis - Linear model:  $Y = a + b \cdot X$

-----  
Dependent variable: Perceived  
Independent variable: Instantiations  
-----

Parameter	Estimate	Standard Error	T Statistic	P-Value
Intercept	6,74012	0,416201	16,1944	0,0000
Slope	-0,000275124	0,000360002	-0,76423	0,4574

### Analysis of Variance

Source	Sum of Squares	Df	Mean Square	F-Ratio	P-Value
Model	1,19884	1	1,19884	0,58	0,4574
Residual	28,7371	14	2,05265		
Total (Corr.)	29,9359	15			

Correlation Coefficient = -0,200117  
R-squared = 4,0047 percent  
R-squared (adjusted for d.f.) = -2,85211 percent  
Standard Error of Est. = 1,43271  
Mean absolute error = 1,21688  
Durbin-Watson statistic = 2,25208 (P=0,2939)  
Lag 1 residual autocorrelation = -0,144403

The StatAdvisor

-----  
The output shows the results of fitting a linear model to describe the relationship between Perceived and Instantiations. The equation of the fitted model is

$$\text{Perceived} = 6,74012 - 0,000275124 \cdot \text{Instantiations}$$

Since the P-value in the ANOVA table is greater or equal to 0.10, there is not a statistically significant relationship between Perceived and Instantiations at the 90% or higher confidence level.

The R-Squared statistic indicates that the model as fitted explains 4,0047% of the variability in Perceived. The correlation coefficient equals -0,200117, indicating a relatively weak relationship between the variables. The standard error of the estimate shows the standard deviation of the residuals to be 1,43271. This value can be used to construct prediction limits for new observations by selecting the Forecasts option from the text menu.

The mean absolute error (MAE) of 1,21688 is the average value of the residuals. The Durbin-Watson (DW) statistic tests the residuals to determine if there is any significant correlation based on the order in which they occur in your data file. Since the P-value is greater than 0.05, there is no indication of serial autocorrelation in the residuals.