# Software Mesurement and Functional Programming

Klaas van den Berg

**PhD Thesis**
**June 23, 1995**
**University of Twente**

**CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG**

**Berg, Klaas Gerrit van den**

**Software Measurement and Functional Programming**
**Thesis University of Twente Enschede**
**ISBN 90-9008251-4**
**Subject headings: software measurement / functional programming**

**This thesis has been approved by**
**Prof. dr. ir. A.J.W. Duijvestijn, University of Twente**
**Prof. dr. N.E. Fenton, City University, London**
**Dr. P.M. van den Broek, University of Twente**

# Abstract

Software metrics have been investigated for the assessment of programs written in a functional programming language. The external attribute of programs considered in this thesis is their comprehensibility to novice programmers. This attribute has been operationalized in a number of experiments. The internal attribute of software which is examined is the structure. Two models for the structure of software have been employed: callgraphs and flowgraphs. The proposed control-flow model captures the operational semantics of function definitions. The objective measurement of the attributes has been supported by tools. The validation of structure metrics has been addressed in certain experiments for programming-in-the-small. The structure of type expressions in functional programs has been analysed in a case study. A simple framework for software metrication proved to be useful. The validation of metrics has been linked with axioms from the representational measurement theory. The control-flow model for functional programs showed its value in the set-up of an experiment regarding the influence of the structure on the comprehensibility. A programming style rule on the use of guards in function definitions has been validated by the findings in this experiment.

# Contents

# Summary

In general, a producer is interested in the quality of his product, whether it is a software package or, for example, a car. There are quality aspects which are important to the user of the product, such as for a car the fuel consumption rate. Other quality aspects are relevant to the technicians who have to build the product or to maintain it: e.g. the ease of assembling certain parts. Furthermore, the producer will be interested in the cost and duration of the production, and the resources needed. Such quality aspects have to be measured to allow a comparison with other products and production processes: a particular fuel consumption rate will be acceptable in certain circumstances.

A similar situation is encountered in the case of software. There are user aspects of quality, for example with respect to the interface and performance, and other aspects related to the programmers who have to design and implement the computer programs. The discipline of software engineering offers methods for the design and production of software. The field of software measurement provides approaches to the quantification of quality aspects of software, related to the product, the process and the resources. An obvious software metric is the size of the program, usually expressed in the number of lines of executable code. But there are many other software metrics, and it is necessary to be able to decide when to use which metric and how. With these metrics, one would like to be able to make an objective assessment of the relative merits of software products and software development methods.

This thesis addresses some issues on the quality of software with respect to the programmers: the comprehensibility of the program code. A lot of time is spent reading and understanding programs in order to remove faults or to adapt the program to changed requirements. Many factors in the program code affect the comprehensibility of the program, such as the language used, the naming of variables, the structure, the indentation, explanatory documentation, the experience of the programmer, and so on.

In order to capture a particular quality aspect of programs, usually a model is built. In such models, certain details in the program are abstracted. The models are used in the definition of software metrics. The models and metrics

have to be validated, for example their consistency has to be established. Furthermore, the metric values can be obtained with the use of tools: i.e. another computer program is used to analyse the original programs. The tools assure a fixed procedure and thus an objective assessment of the quality aspects.

This thesis focuses on the structure of the code, how it is divided into parts - usually called modules - and how the modules are related to each other. This aspect of structure is modelled in a callgraph of the program. Another aspect studied in this thesis is the control structure: the order in which parts of the program will be executed, as prescribed by special language constructs. For this aspect, a control-flow graph of the program is used. The metrics are indicators of the complexity of the structure. These models and metrics are described in Chapters 4 and 5. A tool for the automated measurement of metrics based on these models is described in Chapter 5.

Two classes of programming languages are considered: the 'classical' imperative ones, with languages such as Pascal and Modula-2, and the less common class of functional languages, where Miranda is used as the example. The latter is a very powerful mathematics-like language. These languages are used in the initial programming courses in Computer Science at the University of Twente as described in Chapter 2. One would like to know whether students who learn to program in Miranda write better programs than the students who learn for example Modula-2; and also: are these Miranda programs easier to comprehend than Modula programs? For this comparison, some experiments with certain well-known software metrics are described in Chapters 2 and 3. Some models, the callgraph and the control-flow graph, that are used for imperative languages, are modified for the functional language Miranda as described in Chapter 5.

Once one has obtained metric values, it has to established how they can be used. Do they yield the expected ordering of programs, e.g. with respect to their comprehensibility? Are there threshold values beyond which the programs are difficult to understand, or are very error prone? These questions are part of the external validation. The validation has been carried out in some formal experiments using small programs with first-year students, thus novice programmers. They are described in Chapters 6 and 9. The use of measurement theory in the validation is explored in Chapter 7. It is an open question whether the results of experiments involving novice programmers and small programs can be generalised to expert programmers in the industry working on large programs in teams. Several of these validation issues are raised in Chapter 8 of this thesis.

# Chapter 1

## 1. General Introduction

*How good is functional programming?* This simple question raises many other questions, for example:

- What *is* functional programming: how different is it from the 'classical' imperative programming style?
- Is functional programming *good* for the development of software: can these programs be developed in a shorter time; are functional programs more reliable; are such programs easier to maintain? Another question is whether the functional programming style is *good* in teaching programming: is it easy to learn; do students write better programs?
- How can the *quality* of functional programs be assessed: what are the criteria for reliability and maintainability; how can such attributes be quantified; can this assessment be done objectively?

These questions indicate the two themes of this thesis: *functional programming* and *software measurement*. First, a short characterisation is given of functional programming, and then of software measurement, i.e. that field in software engineering which is directed at the objective quantification of software attributes. Subsequently, an overview will be given of the topics addressed in the thesis and the relation between the chapters.

### Functional programming

Two important programming styles are imperative programming and functional programming. *Imperative programming* - in languages such as Pascal, Modula-2 and Ada - is characterised by the use of variables, commands and procedures. A variable refers to a named storage location whose value (contents) can be modified by means of assignment statements. The value of a variable is determined by its computational history.

By contrast, *functional programming* is characterised by the use of expressions and functions. Expressions are used solely to denote a value. The value of

an expression can be derived from the value of its components. There is a substitutive equality between expressions with the same value. Because of these properties functional languages are called *referentially transparent*: they facilitate formal reasoning about functional programs. Expressions may contain certain 'names' which stand for unknown quantities: different occurrences of the same name in the same context refer to the same unknown quantity. Such names are usually called 'variables', but these variables do *not* vary, as in mathematics (Bird & Wadler, 1988)[1]. Functional programs contain *no side-effects* of any kind. A function call can have no effect other than to compute its result. This makes the order of execution irrelevant - since no side-effects can change the value of an expression. It relieves the programmer of the burden of prescribing the flow of control.

A representative functional programming language is Miranda[2] (Turner, 1986). Some important characteristics of this language are the following:

- It allows *higher-order functions:* functions can be passed as parameter and returned as function result.
- It employs *lazy evaluation*: it is a parameter mechanism whereby an argument is evaluated only if its value is actually required, rather than when the function is invoked.
- It uses a *polymorphic strong typing* system: the type of each expression is checked at compile time, with type variables standing for unknown types.
- It supports the use of *patterns* in the definition of functions.

An example of a program in Miranda is explained in Chapter 5 (section 2) of this thesis; type expressions are described in Chapters 6 (section 3) and 7 (section 2); and patterns in function definitions are described in Chapter 9 (section 2).

Functional programming has its roots in mathematical logic. One of these roots is the lambda calculus developed by Church in the 1930s. Furthermore, in defining functions, recursive equations are used, as formalised by Kleene in the same period. McCarthy (1960) proposed a mathematical basis for computation, which was influenced by the lambda calculus and recursive function theory. This culminated in the LISP programming language, which in its pure form is the first functional programming language. Interest in functional languages increased due to the Turing Lecture by Backus (1978). A major development was the implementation of Miranda by Turner (1979). Other modern functional programming languages are ML, Clean, and Haskell. There are

---

[1] The references are given at the end of the thesis

[2] Miranda is a trademark of Research Software Ltd.

several sources providing an account of the history of functional programming (e.g. Hudak, 1989; Hughes, 1989; Michaelson, 1989) and programming paradigms (e.g. Watt, 1990).

Many claims have been made on the potential of functional programming. Functional programming is expected to deliver an important contribution towards the improvement of software development in its role of executable specifications and prototyping.

## Software measurement

Software measurement is a field in software engineering. Three approaches have been identified in the field of software engineering research (Basili *et al.*, 1991): the formal methods approach, the system building approach, and the empirical studies approach. In the first approach, software development is viewed as a mathematical transformation process. In the system building approach, the emphasis is on finding better methods for structuring large systems. In the third approach, *experimental software engineering*, the emphasis is on understanding the strengths and weaknesses of methods and tools in order to tailor them to specific goals of a particular software project. A cornerstone in this approach is measurement.

A rather formal definition is as follows: *software measurement* is the objective quantification of attributes of software entities: processes, products and resources (Fenton, 1991). Software measurement is needed to gain control over excessive cost of software, low productivity, and poor quality. The original motivations in the early 1970s for deriving software measures were almost entirely managerial, resulting in numerous models for the estimation of software cost and development effort. This also resulted in measures and models for assessing the productivity of personnel during different software processes in different environments. These models had to consider the quality of the software produced, resulting in so-called quality models.

Like measurement in any other discipline, software measurement has to be based on measurement theory (Fenton, 1994). Formally, a *measure* is an objective assignment of a number (or symbol) to an entity to characterise a specific attribute. Measurement is the process of this mapping to numbers. In mathematics, a *metric* is a function defined on a pair of entities x and y such that with respect to a specific attribute, m(x,y) measures the 'distance' between x and y. Unfortunately, there are no generally agreed definitions of metric and measure. In the thesis, the term software measure is used interchangeably with the term software metric.

The interest in the expenditure of human resources on the development and operation of software systems has manifested itself in attempts to quantify software *complexity*. Complexity is perceived as the 'root of all evil' and if only it could be reduced this would bring about attendant reductions in all manner of software evils: excessive development and testing effort, unreliability, and unmaintainability. However, no researchers have yet been able to give an adequate definition of the term complexity (Shepperd & Ince, 1993). Some complexity metrics are considered in this thesis: McCabe's complexity metric in Chapter 3, and other flowgraph-based complexity metrics in Chapters 4 and 5. Software complexity is believed to be reduced by using development methods which provide *structure* to the process and the products. It is an 'axiom' of software engineering that a good internal structure yields a good external software quality (Fenton, 1991).

Many software metrics are described in the literature. Not only do these metrics aim to measure a wide range of attributes but also there are often many irreconcilable metrics all claiming to measure the same attribute such as cost, size or complexity. The reason for this state of affairs is commonly attributed to a general lack of *validation* of software metrics, i.e. ensuring that the metric is a proper numerical characterisation of the claimed attribute.

## Overview

The research in this thesis can be seen as part of experimental software engineering. The approach to software measurement, based on representational measurement theory, is strongly influenced by the work of Fenton *et al.* (1994), especially in the second and third parts of this thesis. Moreover, there is an emphasis on software product metrics and their validation.

Programming in functional languages is considered, as it has been a theme of research at the University of Twente (Berne, Duijvestijn & van der Hoeven, 1985; Joosten, 1989). The research in this thesis originated in an educational setting: an answer was required to the question as to which programming paradigm, imperative programming or functional programming, is the best suited for an initial programming course of first-year Computer Science students. The case studies and experiments have been carried out in this context. The following problems are addressed:

- How can aspects of software quality in the two programming paradigms be assessed and compared using software metrics ?
- How can software in a functional programming language be modelled to capture structural properties of this software ?
- How can software models and metrics be validated in experiments based on measurement theory ?

An overview of the thesis is given in Figure 1.1. There are three parts.

- In part A, the educational *context* of the research on software metrics for functional programming is given.
- In part B, the *modelling* of imperative and functional programs, as used for structure metrics, is explored.
- In part C, the *validation* of software metrics is investigated in some case studies.

In Chapter 10 some general *conclusions* of this thesis are presented.

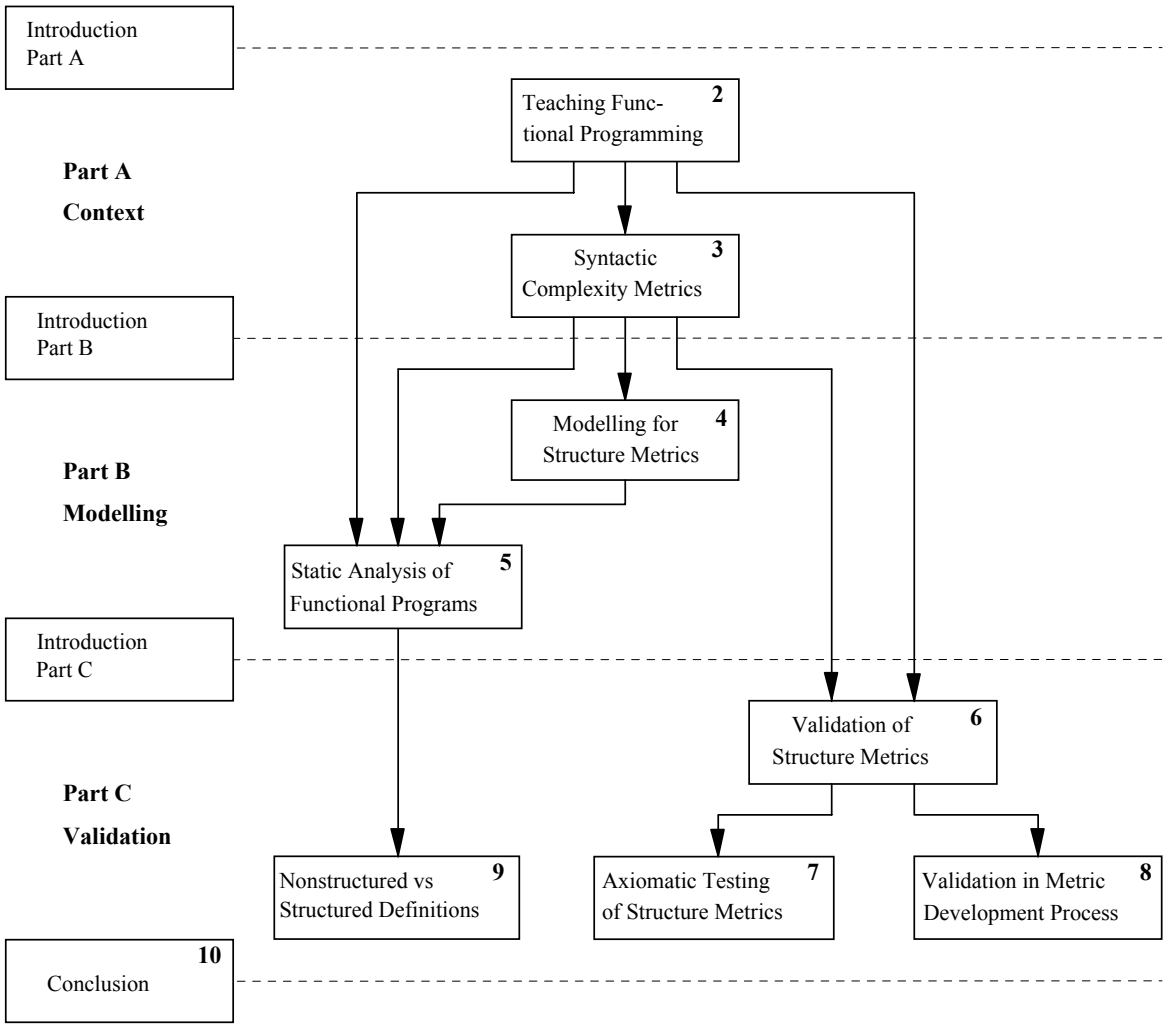



*Figure 1.1 Overview*

The main issues in the chapters in **part A** - the context - are the following:

- In Chapter 2, *Teaching Functional Programming to First-Year Students*, the first experiments with some ad hoc software metrics based on callgraphs are described. The metrics have been used for comparing students' performance in writing functional programs versus imperative programs. An outline is given of the educational context of the experiments (Joosten, van den Berg & van der Hoeven, 1993).
- In Chapter 3, *Syntactic Complexity Metrics and the Readability of Functional Programs*, the 'classical' software metrics of McCabe and Halstead are used in comparing students' programs in an imperative and functional language. A tool for the measurement of these metrics has been based on attributed grammars (van den Berg, 1992).

There are two chapters in **part B** on modelling of software:

- In Chapter 4, *Modelling Software for Structure Metrics*, an alternative to the software metrics used in Chapter 3 is explored: structure metrics based on control-flow graphs for imperative programs. The notion of structure graphs is introduced as an extension to the theory (van den Broek & van den Berg, 1993).
- In Chapter 5, *Static Analysis of Functional Programs*, two graph models are employed for defining metrics for functional programs: callgraphs and control-flow graphs. A metric analyser based on attribute grammars, as in Chapter 3, has been developed (van den Berg & van den Broek, 1995a).

The issues related with validation of software metrics in **part C** are:

- In Chapter 6, *Validation of Structure Metrics: A Case Study*, type expressions in the functional programming language Miranda are modelled and structure metrics are defined. Hypotheses based on internal and external axioms have been tested in experiments (van den Berg, van den Broek & van Petersen, 1993).
- In Chapter 7, *Axiomatic Testing of Structure Metrics*, the testing of axioms has been based on the representational measurement theory (van den Berg & van den Broek, 1994). Deterministic and probabilistic testing of the empirical order has been compared with a theoretical partial ordering of type expressions from Chapter 6.
- In Chapter 8, *Validation in the Software Metric Development Process*, an outline is given of different types of validities in the development process of

software metrics (van den Berg & van den Broek, 1995c). A validation network scheme is combined with the representational measurement approach from Chapter 7.

- In Chapter 9, *Programmers' Performance on Structured versus Nonstructured Function Definitions*, the control-flow model for functional programs from Chapter 5 is used in an experimental comparison of the comprehensibility of structured and nonstructured Miranda function definitions (van den Berg & van den Broek, 1995b).

This thesis consists of a number of published papers. It reflects research over a period of about six years (1989-1995) in the field of software measurement, with an emphasis on software written in functional programming languages. These papers are included in this thesis only with some minor modifications. However, Chapter 8 is a condensed version of the original published text, in order to avoid an overlap with Chapter 7.

# Part A : The Context

## Issues

The initial motivation of the research in this thesis was the search for an objective justification of the decision to introduce functional programming into the first year of the Computer Science curriculum at the University of Twente. The question to be answered was: 'Do students produce better programs when they learn functional programming instead of imperative programming?' This question raised two issues: 'Which criteria can be used to assess objectively the quality of programs', and 'How to compare quality aspects of programs written in different programming paradigms'.

The following quotation characterises the situation seen nowadays in many discussions about software engineering methods and programming paradigms, and also with respect to Computer Science curricula:

> *Much of what we believe about which approaches are the best is based on anecdotes, gut feelings, expert opinions, and flawed research, not on careful, rigorous software-engineering experimentation. (Fenton, 1994)*

At the earliest stage of the research for this thesis, some ad hoc criteria for the assessment of students programs were defined and used in the experiments. The educational setting and the first experiments are described in Chapter 2.

In the subsequent stage, the applicability of software metrics - defined in the literature - was investigated. Two of the most popular, the Halstead metrics (Halstead, 1977) and McCabe's cyclomatic complexity metric (McCabe, 1976), were used in experiments described in Chapter 3. A problem encountered was that these metrics have been defined mainly for imperative languages: therefore, a first task was the definition of the metrics for functional programs. Moreover, a tool based on attributed grammars was developed for the automated measurement of the metrics.

## Program Comprehension

Before developing and using metrics, one must have a clear idea about the purpose of the metric. A Goal-Question-Metric method (Basili & Rombach, 1988) gives a framework for this issue. The goal and some questions have been expressed above. In the research in this thesis, one major issue has been the objective assessment of the comprehensibility of programs. Understanding existing code is one of the more time-consuming tasks in the maintenance of software products. A recent survey of models for code understanding is given by von Mayrhauser (1994). The models are classified as either top-down models or bottom-up models. Top-down models emphasise the nature and structure of domain knowledge and how it is represented in and mapped onto code and documentation information. Bottom-up models build understanding from detail code using control flow and data flow.

Bottom-up models for program comprehension have been used in this thesis. Many other factors influencing comprehensibility - i.e., naming, indentation, typography (cf. Curtis, 1986) - have not been considered. In Chapter 3, the code metrics of Halstead and McCabe have been used as indicators for the comprehensibility of programs in Pascal and Miranda. There are critics of these metrics however (e.g. Shepperd & Ince, 1994), and there is doubt about the applicability of these metrics to program comprehension. This led to the investigation of structure metrics. The modelling of software for structure metrics are described in part B, and the validation of the structure metrics in Part C of this thesis.

As a side effect of our research described above, software metrics have been used in investigating the assessment criteria applied in regular student assignments in imperative programming courses at the University of Twente (Moerkerke *et al.*, 1990). The software quality model developed by Boehm *et al.* (1973) has been utilised to make explicit the criteria applied by lecturers in these assessments. The ranking of criteria obtained in the Moerkerke study provides a basis for a more objective assessment of programming assignments.

# Chapter 2

## 2. Teaching Functional Programming to First-Year Students [3]

*In the period 1986-1991, experiments have been carried out with an introductory course in computer programming, based on functional programming. Due to thorough educational design and evaluation, a successful course has been developed. This has led to a revision of the computer programming education in the first year of the computer science curriculum at the University of Twente. This chapter describes the approach, the aim of the computer programming course, the outline and subject matter of the course and the evaluation. Educational research has been done to assess the quality of the course.*

### 2.1 Introduction

There is a growing interest in lazy functional programming languages such as Miranda and Haskell. It is therefore obvious to investigate whether an introductory course in computer programming can be given in a functional programming language. Because these languages are so new, there are only a few places in the world where functional programming is used in this role.

Until 1991, the introductory computer programming course at Twente was based on imperative languages, that is Pascal and Modula-2. A decision to switch to functional programming is rather drastic, and has been taken with great care. A period of five years has preceded the introduction, in which extensive experimentation and evaluation went together with careful planning and decision making. The functional programming course has been conducted four times in experimental form, with 30 to 40 participants each year. By now, the course has found its definitive form, and has been introduced for all com-

[3] S.M.M. Joosten (Ed.), K.G. van den Berg & G.F. van der Hoeven (1993). Teaching Functional Programming to First-Year Students. *Journal of Functional Programming, 3(1),* 49-65.

puter science students at the start of the 1991/92 curriculum. As a consequence, a large amount of didactic experience has been built upon teaching functional programming as a first language.

In this chapter we want to motivate the choice for lazy functional programming for the introduction to algorithmic thinking. The new programming course is described briefly. The following questions will be answered:

- What is the aim and the subject matter of the introductory computer programming course?
- Why did we choose for this approach?
- Which problems occurred and how did we solve them?

### 2.1.1 Motivation

Research on functional programming has been conducted at the University of Twente from 1982 onwards. Part of this research was directed towards using the functional languages in practice (Joosten, 1989). The idea to introduce our own freshmen to computer programming by means of a functional language dates back to 1986. Although many thought of it as unrealistic, we could think of many reasons why this was a good idea. Some years later many of these reasons still stand. We mention the most important ones.

The concept of algorithm is introduced with a minimum amount of distracting elements such as redundant syntax, details about the order of evaluation, and exceptional situations to keep in mind while programming. Much better than in imperative languages, a functional language enables you to denote appropriate abstractions. Clear and concise programs can be written that express the essence of the algorithm, and nothing more. Such properties have created the necessary room in the course to concentrate on design issues rather than language details.

As imperative programming still dominates this field, we also want to educate our students in an imperative language. We have noted that knowledge of two language families at such an early moment improves the attitude of students towards programming languages. Uncritical language adoration makes place for a more objective attitude.

Functional programming offers a suitable starting point for many fields, such as computer algebra, artificial intelligence, formal language theory, specification etc., and appealing applications are sooner within the reach of students.

### 2.1.2 The students

The course is designed for freshmen students in computer science. Most of them are 18-19 years of age. At high school, all students have taken mathematics and physics classes. Few of them have had previous exposure to computer programming, many have used a computer in one way or another. Since Dutch universities do not have admission examinations, the level of the freshmen cannot be influenced directly by our department.

Most of our students find jobs in business information technology (approx. 50%). The other students find jobs in many different fields, such as process control, science education, telematics, research.

## 2.2 The computer programming course

In this section we describe the structure and the contents of the computer programming course. After a general introduction, each part is discussed in more detail.

The aim of the course is to introduce students to the concept of algorithm and data abstraction for the purpose of designing software on a realistic scale. After successful completion of the course, the student must be able

- to design an algorithm solving a practical problem
- to prove that an algorithm satisfies its specification
- to reproduce and to apply a number of standard algorithms (e.g. backtracking, combinatorial algorithms on graphs, sorting)
- to design software 'in the large' by means of data abstraction (i.e. modularisation)
- implement separate modules and integrate them with modules built by fellow students to create a correctly functioning system

Formal and practical aspects are involved in this. Students must translate a practical problem into an algorithmic notation. At the same time, they must apply formal techniques to prove the correctness of an algorithm and to transform it into an equivalent algorithm. Moreover, students are familiarised with design aspects.

The whole course takes one year and consists of three terms. A term consists of 8 weeks of scheduled activities followed by 4-5 weeks of 'free' time to prepare and take examinations. In this section, the computer programming course is described term by term. Subsequently, the instruction material is addressed.

The form of instruction is similar in each one of the three terms: lectures (8 weekly sessions of 2 hours), tutorials (12 sessions of 2 hours during 8 weeks) and practicals (laboratory assignments) (8 weekly sessions of 4 hours). On the average a student spends about 50 hours on self study, involving homework, exam preparation, reading, etc. The practicals are obligatory. A student spends about 125 hours in total during each term.

### 2.2.1 Functional Programming

In the first term the students get acquainted with algorithms expressed in Miranda (Turner, 1986). The subject matter covers most of Bird and Wadler (1988). At the end of this term the students have written many different algorithms in a functional language, the complexity of which is comparable to quicksort, tree traversal, folding with minor pitfalls and the like. Also, they have designed and built a few larger programs of a more complex nature. Students have shown that they can define one function in several different ways, for example recursively, with list comprehension or with standard functions. Also, students have made several proofs based on structural induction. They have diagnosed errors in given definitions. Finally, they have translated a number of practical problems to suitable data structures with accompanying functions. The remaining skills have been demonstrated in practical work. Most of these skills are tested by means of an examination.

In the tutorials, many small exercises are done to make the theory operational. The tutorials offer a lot of practice in theoretical issues, such as proof techniques. Examination results show that students cope with proofs well.

In the laboratory students solve realistic problems. The first sessions comprise small exercises that are intended to familiarise students with the language. These exercises are done individually. Solving 'realistic' problems starts about halfway the first term. From that point students work in pairs. There is supervision (1 supervisor per 12 students) to prevent a pair of students getting stuck for too long. Otherwise, they can just carry on and solve their own problems.

The first problem solving assignment is one in which students have to design the contents of a file containing information about a given situation. This file is built as a list of n-tuples, and contains (depending on the concrete assignment of each student) family relations, football results, ingredients for cooking, and so on. The students have to write a program to provide answers to questions like: which teams have lost a football match at home? Such problems can usually be solved with a 'one-liner' that uses a list comprehension:

```
lost          :: [footballresult] -> [team]
lost results =  [home|(home,visitor,scoreH,scoreV)<-results
                     ; scoreH < scoreV]
```

Usually it takes a while for students to discover that the problem can be solved in such a simple way. Each student writes a program to answer approximately four of such questions. The student has to create an input file with test input, and make the whole thing work. This assignment is illustrative for the other assignments. Other assignments include a modification of the calendar program from Bird and Wadler (1988) according to a given requirements specification, interactive programming and writing a program to manipulate tree structures. By means of the lab assignments, students develop a reasonable experience in problem solving and programming. Compared with the 'old' curriculum, students solve more problems of a more complicated nature.

The reader may appreciate that we disagree with the popular belief that functional programming would be more theoretical (than imperative programming).

### 2.2.2 Imperative Programming

In the second term students learn imperative programming. The students must learn to write a good, conventional style, imperative program. However, in presenting the material we benefit from the abilities acquired in the first term. One reason to expose students to a second language early is to prevent them from acquiring unmotivated preferences for 'their' language.

We have chosen Modula-2 instead of Pascal, because Modula-2 offers standardised support for modularisation. Abstraction being a major issue in this course, it is desirable to have a language that supports modularity well.

The imperative course must ensure that the skills of students with respect to imperative programming are at least equal (if not better) than the skills of students in the old curriculum. In that sense, this is an ordinary programming course. However, the approach is different because one can take advantage of the functional programming skills acquired so far. For example, recursion is not treated as a separate subject, but is used without introduction. Procedures as parameters are used from the very beginning, because students are used to higher order functions. Function procedures are used frequently. Functions yielding composite types as result are supported, although this is not a standard Modula-2 facility. Standard operations, such as arrays, lists and trees, are offered in reusable modules. These operations correspond, as much as possible, with operations already known to the students from the first term. By

using the built-in operations, students are trained to solve problems at a higher level of abstraction. They adopt this style in the way they define their own functions.

As mentioned, abstraction is a big issue in the second term. Students must learn to abstract from concrete aspects, and find the right abstraction level to express a problem. Either they use or they define the proper procedures to reach that level of abstraction. Students learn to lift a set of standard operations to a new set of standard operations that allows them to solve their problem adequately. The concept of abstract data type is treated in Miranda and Modula-2 in parallel.

The difference with functional programming is emphasised by reasoning about programs in conventional state semantics. Students are taught to reason about programs in terms of state assertions (Floyd-Hoare logic). Students learn to consider the control flow explicitly, and to make decisions about the representation of data. These issues (control flow and data representation) remain implicit in the functional world.

### 2.2.3 Programming techniques

In the third term, functional and imperative programming are used in the context of software design. The subject matter in this term consists of two parts. The first part is a treatment of programming techniques, standard methods and categories of problems. Students learn to use backtracking and branch & bound techniques, pattern recognition and parsing, finite automata in dialogue construction, sorting and shortest path algorithms. For each of these topics the same aspects are treated:

- standard techniques and algorithms;
- complexity considerations;
- relation between functional and imperative programming;
- data abstraction;
- documentation.

In the second part, students carry out a programming project. The students are confronted with a system that consists of 10 modules. A prototype system, written in Miranda, is available for experimentation purposes. This system is written entirely in the functional realm. The students learn about the system by studying it, and making their own version of the dialogue specification.

Then they get a partial implementation of the same system, written in Modula-2. Two modules have to be added to this system in order to complete it. Certain data structures are implemented 'invisibly', so students are confronted directly with the consequences of abstract data types.

Finally, the students integrate their own modules with the modules of other students, yielding a complete system written by several people independently. Section 2.4 on the Programming project contains a detailed description of this assignment.

This approach has advantages over letting students make a full program from scratch. In this situation students have to delve into existing software, which confronts them with important issues like maintainability, the role of specification etc.

After completion of this third term the student is able:

- to specify a practical problem in the form of an initial algorithm;
- to transform the initial algorithm to an efficient algorithm;
- to convert this algorithm to an imperative implementation;
- to document the design process.

### 2.2.4 Instructional material

Much effort has been paid to the development of instructional material. Not only have we looked carefully at the textbook, but we have also paid a lot of attention to other kinds of written material, to support students as well as instructors.

As textbook for the first term we have chosen *An introduction to functional programming* by R. Bird and P. Wadler (Bird & Wadler, 1988). Although we do not advocate it for self study, this book has about the right mix of practice and theory. We feel it is important to use a textbook that does not deal with implementation of functional languages.

An alternative (at the time) would have been Wikström (1987). The latter has a less formal approach, and therefore it has been considered as less suitable for this term. A more recent introductory textbook is by Holyer (1991).[4]

In the second term, students use lecture notes on programming in Modula-2 (with previous knowledge of functional programming) (van den Berg, 1992a). The book written by Koffman (1988) has been chosen as a reference. In the third term we do not use another book, but rely on our own material and the books already mentioned.

A book of exercises has been composed, partly with worked out solutions (op den Akker *et al.*, 1992a). This material is a student's companion to Bird and Wadler (1988). Scripts have been worked out for all lectures, tutorials and lab sessions, making explicit the aim of each session (op den Akker *et al.*, 1992b). This is a teacher's manual to the course. The students obtain a copy of the

[4] At the time of publishing this thesis we refer to Myers, Clack & Poon, 1993, 1995; Ullman, 1994

transparencies used in the lectures, serving as a supplementary text. These texts are all in Dutch.

In the first term students program in Miranda (Turner, 1986) in a UNIX environment. The second term the students use Modula-2 in a MS-DOS environment, specifically using TopSpeed Modula-2 (Jensen, 1988). In the third term both language systems are used.

## 2.3 Evaluations

The development of this course started in August 1986. A five-year plan was made for extensive experimentation, leading to the definitive introduction in 1991. In 1987/88 we started teaching with a group of 24 volunteers, out of some 120 freshmen of the faculty of computer science. We found that this group had scored 10% (on average) higher on the mathematics and physics examinations in high school. Also, these volunteers scored about 10% better than their peers in the other subjects taught in the first year at the university. Apparently, this group was far from representative. Therefore, this course could be used only for trying out the material. Comparative studies could not be done until the following year. In 1988/89 we composed two representative groups totalling 48 students. In 1989/90 we proceeded with two groups (in total 40 arbitrarily chosen students) in this new computer programming course. In the last preliminary year, 1990/91, the course has been executed in its definitive form on two groups of 34 students in total. Over these years, the functional programming course has evolved considerably. Student results and appreciation and learning speed have improved considerably. Also, the major part of the old imperative curriculum is covered in the second and third term of the course. From 1991 onwards, all students (up to 120 per annum) are taking this course.

### 2.3.1 Observations

Evaluation of the courses has been performed in close co-operation with the Educational Research Centre of the university. Regular discussions have been held between staff and students, instructors and educational experts and the people carrying out the actual teaching. After each term, questionnaires have been used to measure opinions and attitudes of students. In the first year of experimentation (1987/88) exact time measurements have been performed to assess the time spent by students. In the other years a detailed estimate of time spending was asked in the questionnaires.

The students judge the course to be not very difficult compared to the other courses in the first year. The rather formal textbook in the English language,

which is not native to our (Dutch) students, is not experienced as a problem. The time expenditure is in good agreement with the norm (and is even favourable compared with the programming courses in the old course). In general the students find the course pleasant and useful.

Another source of information, albeit 'soft', is the experience and the impressions of participants (both students and tutors). From the open remarks on the questionnaires, the discussions with students and colleagues and the performance of students at the examinations, we have become convinced that students can cope with the higher level of abstraction. We think that this is an improvement over the classical programming education. The ability to make a program work by means of trial and error is less useful to students than it used to be.

Since this is a freshman course, the department is interested to know how this course separates the better students from the poor performers. In the new course, students are selected much more on their ability to make abstractions. In the old course, we have the impression that smart programmers with insufficient abstraction ability would sometimes pass only because they can make programs work.

### 2.3.2 Problems

Over the years, we have encountered problems that have to do with the way in which functional programming is taught. Such problems were foreseen. In 1986, functional programming was taught only as a facultative subject for students with reasonable experience in imperative programming. Not much instructional material was available in 1986 (cf. Bailes 1989; Savitch 1989), and similar courses are mostly of a more recent date. So, the course and the material have been developed from scratch. Teaching it to students with no previous exposure to programming was considered risky, because functional programming has a reputation of being difficult. The importance of a freshman course for the entire (4-year) curriculum is such that a lot of time was needed to experiment and introduce the course. This created the opportunity to analyse educational problems properly, and think of good solutions. Three of these problems are discussed in the following three sections.

#### 2.3.2.1 Priority and associativity

Many problems in understanding Miranda expressions in the first courses were connected with the priority and associativity-rules and the placing of parentheses, especially with the 'invisible' function application operator. Students have a hard time getting used to the way operators interact with func-

tion application. For example $f\,.\,g\,x\,y$ is often read as $(\,(\,f\,.\,g\,)\,x\,)\,y$, whereas it really means $f\,.\,(\,(\,g\,x\,)\,y\,)$ .

To solve this problem we have introduced special exercises to train this ability. Furthermore, we use the @-symbol during the first weeks if we need an explicit denotation for function application. This helps when students have problems with the implicit presence of the application operator. For students who grasp the idea right away, we use normal notations only. Furthermore, we tend to draw syntax trees to make the parsing of an expression explicit. The role of parentheses is explained in connection with these trees.
So, $f\,.\,g\;\;x\;\;y$ is written as $f\,.\,g\,@\,x\,@\,y$ or with parentheses $f\,.\,(\,(g\,@\,x\,)\,@\,y)$ . The corresponding syntax tree is drawn in Figure 2.1.

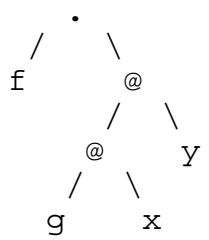



*Figure 2.1 Syntax tree*

#### 2.3.2.2 Type expressions

In the beginning we had much trouble with Miranda's types. Students made many mistakes, both in the laboratory and on paper. There are several aspects of the errors made with type expressions. Before considering solutions to this problem, we have made an inventory of these mistakes. The following categories of mistakes were identified:

**Understanding given type expressions**
- The function arrow is given the same associativity as the function application: $a \to b \to c$ is read as $((a \to b) \to c)$
- The main structure of the type expression is not recognised:
  e.g. $a \to b \to c$ is interpreted as the type of a 3-argument function.

**Giving the type of a specific function**
- No parentheses are placed around arguments that are functions
- Functions with more than one argument are not recognised
- The result type is replaced by some type expression of the right hand side of the definition or omitted at all

- Too many restrictions are placed on types, e.g. all types are *num*.
- Too few restrictions are placed on types, e.g. all types are polymorphic type variables and not bound to specific type.

**Miscellaneous errors**

- Errors in understanding type error messages are mostly due to a wrong interpretation of terms used in these messages: cannot unify, cannot apply, cannot identify. Frequently, students do not use the actual content of the error message, but solely the indication of the place where something is wrong.
- Errors due to naming conventions, such as *xs* and *ys* for lists. Several students think that the computer derives the type [∗] based on these names.
- Type expressions are mixed with ordinary expressions, like this for example: *last* [∗] = *head* (*reverse* [∗])

The problems with types were also clearly visible in the evaluation results in the first two years of experimentation (van den Berg & Pilot, 1989). 'Giving the type of a function' was among the first three subjects in the list of ten most difficult issues.

Major adjustments of the course have taken place, based on these observations. Firstly, we relaxed the requirement that a student should be able to derive the type from an expression. Now we require that the students can write (as opposed to derive) the type of their own definitions. In order to give the necessary practice and to advocate good programming style, we insist that the type is given explicitly with every definition. Interpreting error messages remains a problem. Phrases like ‘cannot unify’, ‘cannot apply’ and ‘cannot identify’ are explained in an introductory practical assignment, which helps a little. As a result of all these measures, the topic of typing has disappeared from the top ten of difficult issues.

#### 2.3.2.3 Computational model

In the experimental phase of the courses, the students received the functional and imperative programming courses in parallel. Interference of both courses has been observed, especially in the case of the computational model. The computational model for functional programming is based on rewriting and lazy evaluation, for imperative programs on memory states and state transitions. Some errors occurred because students used the imperative model in the functional programming domain:

- Some students thought that the definitions in the script should have a particular order: ‘otherwise the value of a variable is not known’.

- They assumed changes in the value of variables by function application; e.g. taking the tail of a list *ys* would change the list, in other words they expected the effect of an assignment *ys* := *tail* (*ys*).
- Some felt the need to store intermediate results, otherwise these results would be lost, e.g. they wanted to save the original list before calculating the last element with *last xs* = *head* (*reverse xs*).

Apparently, some of the misconceptions are induced by imperative language use in the functional domain: e.g. names like *take*, *drop*, *remove*, *filter* could imply some (side-) effects on the argument of the function. This interference nearly completely disappeared after the imperative programming course has been placed after the functional course.

### 2.3.3 Functional versus imperative programming [5]

One educational experiment has been conducted which is of particular interest: an experimental comparison of the programming abilities of students who have been first exposed to computer programming by means of Miranda versus those who followed conventional programming education based on Modula-2. Since the functional programming course was given to a part of the total first-year population, we had an ideal opportunity to do comparative research (van den Berg, Massink & Pilot, 1989).

In the first experimental design (see Table 2.1), there are two equivalent groups of first year students in computing science: the mathematics and physics grades were used as pre-test. The two groups received different treatments: the functional programming course (*FP*) or the imperative programming course (*IP*). Two experimental conditions were provided: *time pressure* and *no time pressure*. The programming abilities were tested after the course: in the post-test the students received a number of assignments on different aspects of programming. These tests differed only in the programming language used. The number of students *N* for each condition is given in Table 2.1.

| | | Post-test | |
|---|---|---|---|
| Group | Treatment | Time pressure | No time pressure |
| FP | Functional | N = 15 | N = 14 |
| IP | Imperative | N = 11 | N = 10 |

*Table 2.1 Number of students N in experimental design 1*

[5] The text and tables in this section of the thesis differ slightly from the original published text

Several aspects of the programming ability of students have been tested in the given assignments. These abilities are the following: to specify a function; to write comments to a function; to write the type of a function; to identify semantic equivalence between different program constructs; and to use structured data types.

The F-statistic has been used to test differences between the means of these quantities. No significant differences ($\alpha = 0.10$) have been found for each of these abilities in both conditions.

In a subsequent experiment, two assignments were offered with one condition only (no time pressure). The first of these assignments (Assignment 1) comprised the modification of an existing program. Assignment 2 was the design and implementation of a new program for a given specification. The experimental design, with the number of students per condition, is given in Table 2.2.

| | | Post-test | |
|---|---|---|---|
| Group | Treatment | Assignment 1: Modification | Assignment 2: Design + Implementation |
| FP | Functional | N = 11 | N = 8 |
| IP | Imperative | N = 9 | N = 10 |

*Table 2.2 Number of students N in experimental design 2*

For the modification assignment (Assignment 1), the following four quantities were determined: the number of new local functions; the number of new global functions; the number of modified functions; and the percentage of students who modified the main function.

| | | Statistics | | |
|---|---|---|---|---|
| Quantity | Group | Mean | F | p |
| # New local functions | FP | 0.4 | 4.6 | .045 |
| | IP | 0.0 | | |
| # New global functions | FP | 1.5 | 16.6 | .001 |
| | IP | 0.3 | | |
| # Modified functions | FP | 0.1 | 17.0 | .001 |
| | IP | 0.8 | | |
| Modification main function | FP | 82% | 9.9 | .006 |
| | IP | 22% | | |

*Table 2.3 Results for modification assignment*

Again, the F-statistic has been used to test differences of the means of these quantities. The results (means, F-value and p-value) are shown in Table 2.3. The differences are significant ($\alpha = 0.05$).

For the design and implementation assignment (Assignment 2), a program call graph has been derived for each solution. The following five quantities were determined: the number of user defined functions in the graph; the number of levels in the graph (transformed to a tree by removing recursive calls); the maximum number of functions directly called by another function; the number of functions identified in the design; and the coverage, i.e. the percentage of design functions recognisable in the implementation.

The results are shown in Table 2.4. There was no significant difference found between the two groups on the maximum number of functions directly called by another function and not on the number of functions identified in the design. The differences for the other quantities are significant ($\alpha = 0.10$).

| | | Statistics | | |
|---|---|---|---|---|
| Quantity | Group | Mean | F | p |
| # User defined functions | FP | 6.9 | 16.4 | .001 |
| | IP | 2.6 | | |
| # Levels in program graph | FP | 1.8 | 10.3 | .005 |
| | IP | 0.7 | | |
| Coverage design/program | FP | 92% | 3.9 | .064 |
| | IP | 57% | | |

*Table 2.4 Results for design and implementation assignment*

From the results for the modification assignment it can be concluded that students in the FP-group (the functional programmers) introduced significantly more new functions to accomplish the required modification than students in the IP-group (the imperative programmers). The latter realise the required new functionality by changing the existing program at the lowest level code. At the main level, this change is less frequently visible for these students than for students in the FP-group.

From the results for design and implementation assignment it can be concluded that the correspondence between design and program is significant higher for students in the FP-group than in the IP-group. Students in the IP-group use more levels of abstractions with more functions in their programs than students in the FP-group.

Although it is rather subjective to derive the quality of programs from criteria used above, it could be argued that the results on these experiments give

evidence that students in the functional programming group produce programs with a better structure than students in the imperative programming group.

## 2.4 Programming project

In this section we describe a programming project that is conducted at the end of the third term. It serves the purpose of illustrating the type of assignments students do. It allows the reader to form an idea of the level obtained at the end of the first year. The description starts with a discussion about the educational and organisational aspects of the assignment. After that, we show some technical details.

### 2.4.1 Organisation

In the second half of the third term, students work on a larger assignment. Each student spends 16 hours in the laboratory on this assignment, divided in four sessions of four hours each. The work is done in pairs. During the first session, the students are confronted with a Miranda prototype of the system. In the end, this system will be built by those students in Modula-2. They are supposed to experiment with the prototype, to carefully analyse its behaviour, and to present the results of their analysis in the form of an external specification of the system.

The next two sessions, eight hours in total, are devoted to the implementation of parts of the system. The students will have to integrate their work with the work of others, so they realise the importance of sticking to the specification, test thoroughly and remain on schedule. The external specification is used as a starting point. Students do not use the external specification they built themselves, for that was handed in earlier. Instead, they all use the same specification provided by the supervisor. This annihilates the risk of delay for students who have had trouble making the specifications. In this way, students skip part of the design trajectory. What they are supposed to do here, is just to fill in the design. This requires a passive understanding of the structure of the system, the skills required to come to a satisfactory system design by themselves are not taught nor trained in this course.

The final session is for integration of system parts. Four couples of students will now merge their material into one complete and working system.

### 2.4.2 Railway information system

The students work on a restricted railway information system. In this section we give an overview of the system. We hope to give the reader a feeling for the

kind of application we are talking about. This description is not intended to be complete.

The railway information system computes the price of the cheapest ticket for a given journey, taking into account possible reduced fares for reduction-pass holders, group tickets, etc. The system has two important aspects. One is the correct functionality: it should collect the proper data on a journey, and correctly compute the price of the ticket from these data and the information it has stored on the costs of various kinds of tickets. The other aspect is its user interface. It should facilitate the presentation of data on a journey, and handle errors in the input in a clear and understandable way. Any user should be able to consult the system, without much explanation.

```
ticketPriceSys :: aTable -> aTicketStream -> aPriceStream
ticketPriceSys ticketPriceTable
  = map (ticketPrice ticketPriceTable)
ticketPrice    :: aTable -> aTicket -> aPrice
ticketPrice table ticket
  = bp        , if n=1
  = min2 np gp, otherwise
    where
     bp   = basePrice
                table
                dist
                (sinOrRet ticket)
                (class ticket)
                (fulOrRed ticket)
     dist = distance table (dep ticket) (dest ticket)
     np   = bp * n
     gp   = groupPrice table n
     n    = numberOfP ticket
```

*Table 2.5 The railway system function*

The two main requirements of the external specification are that it defines the functional behaviour of the system, and that it defines the form and the nature of the interaction between user and system. The functional behaviour is described by means of abstract data types. Students have to realise which operations are necessary and have to worry about the exact content of these operations. There is a close relation between the abstract data types in the Miranda program and the modules in the Modula-2 implementation. The interaction between the user and the system is described by regular expressions over some alphabet of events. Both elements in this specification lead to a precise formu-

lation of pre- and postconditions, which is useful when designing the Modula-2 code.

To conclude this account of the railway information system, we present the main piece of the functional program: the system function *ticketPriceSys* (Table 2.5). This code was made by a student by way of specifying the interactive behaviour of the system at the global level. Students are expected to produce such code in the external specification they make in the first session of this assignment.

The presentation of these functions presupposes the introduction of types and functions, which can be done at an abstract level (Table 2.6). Somewhere the specification must show that the following (abstract) types and functions are involved, given the types: *aTable*, *aTicket*, *aPrice*, *aNumberOfP*, *aDistance*, *aStation*.

```
aTicketStream == [aTicket]
aPriceStream  == [aPrice]
aWay        ::= Single | Return
aClass      ::= First  | Second
aFare       ::= Full   | Reduced

dep         :: aTicket -> aStation
dest        :: aTicket -> aStation
sinOrRet    :: aTicket -> aWay
class       :: aTicket -> aClass
fulOrRed    :: aTicket -> aFare
numberOfP   :: aTicket -> aNumberOfP
distance    :: aTable  -> aStation  -> aStation -> aDistance
basePrice   :: aTable  -> aDistance -> aWay -> aClass ->
               aFare   -> aPrice
groupPrice :: aTable  -> aNumberOfP -> aPrice
```

*Table 2.6 The types of the functions*

### 2.4.3 Experience

In itself, the assignment is not a difficult one. Most of the students will succeed in the integration of their own part with those of their fellow students. However, it turns out to be very illuminating in several aspects. It confronts students with their own mistakes, their lack of thorough testing, the problems caused by ill-structured code, and so on. It clearly shows the necessity to stick to specifications, if you want your part of the system to co-operate with other parts. It shows that it is most useful to test parts of the system separately and

thoroughly before they are put together. And finally, it confronts students with the problems of version management: it happens more than once that they start integrating versions of modules which are not the final ones, e.g. because they contain material which was put there solely for the purpose of testing.

It is worthwhile to observe the students as they work. Some of them sit down at the keyboard and do 'trial and error' development. Others sit down and think everything through, starting from the (given) Miranda prototype and the specification down to the Modula-2 code. During the integration session, the modules produced by the former students usually contain the problems, whereas the modules of the 'thinkers' often operate flawlessly.

The role of functional programming in this assignment is restricted. It is true that in this assignment for the first time students see a larger piece of software, which performs a useful function, and which is written in Miranda. But they do not themselves develop program of comparable size in Miranda. The skills in functional programming are used to capture the essence of the functional behaviour of a system.

### 2.4.4 Role of functional programming

There are two basically different ways of using a functional program as an intermediate step towards an efficient imperative implementation. One way is by doing program transformations and the other way is by programming and justification. The first 'method' contains the following steps:

1. to write the functional program
2. to transform this program by means of correctness preserving steps until the program is fully tail recursive without using intricacies.
3. to rewrite the result (mechanically) into an imperative language.

The second way is much more informal. A functional program is written and used as a formal specification. The imperative program is developed 'as usual', in which the experience of making the specification makes the big difference in the quality of the resulting product. Proof techniques must be used to get an *a posteriori* justification for the program.

We made the choice for the second way on a rather practical basis: The transformational approach requires a greater skill and education than the second approach. We have educated students in program transformations to a level where they can make proofs. There is no room in the first year program to enhance these skills further to a level in which transformational programming becomes feasible. In the current situation, these skills do not belong in the first year. This motivates our choice for a 'limited' importance of functional programming in the design of software.

Students who have built (their share of) the railway information system report that they appreciate what they have learnt: to capture the functionality of a system in a concise functional specification that fits on the back of a business-card. At the same time they find it useful to have the experience of successfully integrating their work with the work of so many other students. Students appreciate the value of modules and abstract data types in software design. This is an issue that is taught better by experience than in the classroom.

## 2.5 Conclusion

The design and implementation of a new computer programming course was completed successfully within the original time constraints.

Based on research done in this period, we conclude that the quality of the introductory computer programming course has improved. The students learn to handle abstraction as a design tool and are able to describe their problem formally. The skills in formal manipulations have improved. Students solve more problems that are also more challenging. Because imperative programming is still taught, no problems need to be expected with respect to the connection to other (existing) parts of the curriculum. Passing or failing of students depends more on their abstraction skills and less on their coding abilities. Appreciation of students is very high.

# Chapter 3

## 3. Syntactic Complexity Metrics and the Readability of Functional Programs [6]

*This chapter reports on the definition and the measurement of the software complexity metrics of Halstead (1977) and McCabe (1976) for programs written in the functional programming language Miranda. An automated measurement of these metrics is described. In a case study, the correlation is established between the complexity metrics and the expert assessment of the readability of programs in Miranda, and compared with those for programs in Pascal.*

### 3.1 Introduction

Computer programs are written in order to carry out the solution to a problem by a computer. During the construction of programs there is a continual need to read and understand the text of programs: for the further development of the program to meet new specifications, for the modification of programs to correct errors, or to consider programs or program parts for possible reuse. The programs can be written in various programming languages, which may differ in their expressive power. The expressive power of a language is reflected in the ability to write programs that are both succinct and understandable (Fleck, 1990). Understandability or readability will be defined as the extent to which the function of the program and its components are easily discerned by reading the program text (Boehm *et al.*, 1978). We will consider the readability of programs written in the functional programming language Miranda (Turner, 1986) and the imperative programming language Pascal.

---

[6] K.G. van den Berg (1992). Syntactic Complexity Metrics and the Readability of Programs in a Functional Computer Language. In: F.L. Engel, *et al.* (Eds), *Cognitive Modelling and Interactive Environments in Language Learning*. NATO Advanced Science Institute Series, Berlin: Springer, 199-206.

The basic difference between functional and imperative programming languages lies in the hiding of the computational model (Petre & Winder, 1990). The imperative model incorporates the Von Neuman machine characteristics in the notions of assignment, state and effect. A characteristic of this language class is the explicit flow of control, e.g. sequencing, selection and repetition. In assignments, the value of memory places denoted with variables is changed during program execution. This model of operation by change of state and by alteration of variable values is also named 'computation by effect'. The functional model is characterised by 'computation by value'. Functions return values and have no side-effects, and expressions represent values. There is no notion of updatable memory accessible by instruction. The program consists of a script with a number of mathematical-like definitions and an expression that must be evaluated. Functions can be passed as arguments to other functions and can be the result of a function application (higher-order functions). These programs possess the property of referential transparency, which means that in a fixed context the replacement of a subexpression by its value is completely independent of the surrounding expression. Therefore functional programming is more closely related to mathematical activities (Bird & Wadler, 1988).

There is a growing interest in functional programming languages, because of their expressive power and the possibility to reason about correctness of programs. There are claims on the readability of functional programs, for example: 'In many cases the functional programming style yields more elegant and comprehensible programs than the imperative programming style' (Springer & Friedman, 1990) and 'Functional programming leads to programs which are exceptionally clear and concise and to the prospect of greatly increased software reliability and development speed' (Bailey, 1990). In a case study on the productivity of programming in a functional programming environment, some of these claims have been confirmed (Sanders, 1989).

Moreover, the learning of programming in a functional programming style should have advantages over learning in an imperative style (Springer & Friedman, 1990; Bailes & Salzman, 1989). At the University of Twente, first year students in Computer Science receive a course in functional programming, which forms the base of the programming curriculum (Joosten & van den Berg, 1990). An important aspect of teaching programming is to give feedback to the novice programmers on the readability of their programs and intermediate products during the design process. This readability can be assessed by teachers. It is also possible to measure internal attributes of programs based on the syntax of the program text that correlate with the readability of programs. These values can be used in feedback to the students, which may be interactively during the program development. The measures of

software attributes are usually called software metrics and are described in the following section. In a subsequent case study we will report on the correlation between readability and the software metrics for programs in Miranda, and compare this with the correlation for programs in Pascal.

## 3.2 Software Metrics

Software metrics are used to assess the process of the construction of software, the products of this process, and the use of human and machine resources (Conte, Dunsmore & Shen, 1986; Fenton, 1991). We will focus our discussion on the use of human resources in the interaction with software. This attribute of software is referred to as the psychological complexity (Curtis *et al.*, 1979), and we restrict ourselves to the readability or comprehensibility of the program text. The computational complexity, i.e. the efficiency of the use of the machine resources (time and memory space), will not be considered here.

There are several software metrics for the static syntactic complexity of program text described in the literature. We will use the software metrics based on Halstead's Software Science (1977) and the cyclomatic complexity number of McCabe (1976). The study of Curtis (1981) shows empirical evidence that the complexity metrics of McCabe and Halstead relate to the psychological complexity, as expressed in the difficulty in understanding and modifying software. The relation between syntactic complexity and cognitive complexity has been investigated by Khalil & Clark (1989). Shen, Conte & Dunsmore (1983) give a critical review of Software Science. They show that the effort E, as defined by Halstead (see below), correlates with the understandability of programs. However, the psychological aspects of the Halstead metrics have been criticised (Coulter, 1983).

The software metrics of Halstead and McCabe have been applied mainly to programs written in imperative programming languages. We have derived these metrics for the functional programming language Miranda. The definition of these metrics for this language will be given in the section below, followed by a description of an automated measurement of the metrics.

### 3.2.1 Halstead and McCabe Metrics

Halstead (1977) proposed in his theory of Software Science that some useful measures for computer programs can be derived from four basic metrics: the count of unique operands and operators, and their total frequencies. A symbol in a program that specifies an action is considered an operator, while a symbol used to represent data is considered an operand. Among the derived metrics

are the program volume, the level of implementation and the programming effort.

The basic metrics are defined as the number of unique operators $\eta_1$, the number of unique operands $\eta_2$, the total number of operators $N_1$, and the total number of operands $N_2$. The vocabulary of a program is defined as $\eta = \eta_1 + \eta_2$, and the length of a program as the total number of tokens $N = N_1 + N_2$. The volume (size) of a program is defined as $V = N \times \log_2 \eta$. The potential volume V* of a program represents the size of the program in its most succinct form. Halstead showed that $V^* = (2 + \eta_2^*) \times \log_2 (2 + \eta_2^*)$, where $\eta_2^*$ is the number of different input/output parameters. The program level, or the level of implementation, L is the ratio of the potential volume V* and its actual volume V, i.e. L = V* / V. The effort to generate a program is defined by the relation E = V / L. Other quantities have been defined and relations between these quantities can be obtained by algebraic manipulation.

McCabe (1976) developed a measure of software based on the decision structure of a program. The program is represented as a graph G with a unique entry and exit point. The edges represent branches caused by a decision, and the nodes represent a piece of code. The metric counts the number of linear independent paths through a program and is also called the cyclomatic complexity number. This metric is related to the difficulty of testing a program. The cyclomatic complexity number is v(G) = e - n + 2. The number of edges in the graph is e and the number of nodes is n. It can be shown that v(G) is equal to the number of decisions in a program plus one, provided that all decision nodes have outdegree 2.

### 3.2.2 Metrics for Pascal and Miranda

The Halstead and McCabe metrics for the imperative programming language Pascal have been used as described by Conte (Conte *et al.*, 1986). All variables and constants are counted as operands. The operators are the arithmetic operators, relational operators, boolean operators, procedure and function counts, and multiple entities BEGIN END, IF THEN, IF THEN ELSE, WHILE DO, FOR DO, REPEAT UNTIL, CASE END, RECORD END, ARRAY OF, SET OF. The decision count for the cyclomatic complexity number is based on the occurrence of the symbols WHILE, FOR, REPEAT, IF, OTHERWISE, AND, OR, PROCEDURE, FUNCTION, PROGRAM, CASE and commas in the CASE statement.

We have developed the Halstead and McCabe metrics for the functional programming language Miranda. (Similar metrics have been established by Samson *et al.* (1989), for the languages Hope and OBJ.) We consider constants and all identifiers that are not operators as operands. The operators are the

standard operators (including the list operators for list construction ":", list concatenation "++", list difference "--" and selection from a list "!"), the function name in the right hand side (RHS) of function definitions, parameters in a compound definition, delimiters in expressions ("[ ]", "( )","..",";",","), and in function definitions the symbols "otherwise" and "where".

The number of decisions has to incorporate the use of patterns in arguments of function definitions, in compound definitions and in list comprehensions (i.e. expressions from the Zermelo Frankel set theory). A function definition consists of a left-hand side (LHS), the symbol "=" and the right-hand side (RHS).

The cyclomatic complexity number of a script is equal to 1 + the sum of cyclomatic complexity numbers of each function definition. The cyclomatic complexity number of a function definition is equal to the sum of the pattern complexity of the LHS, the number of guards in the RHS, the number of logical operators in the RHS, the number of filters in a list comprehension in the RHS, and the pattern complexity in a list comprehension in the RHS. The pattern complexity is equal to the number of identifiers in the pattern, minus the number of unique identifiers in the pattern, plus the number of arguments that are not identifiers.

We illustrate the determination of these metric values for a small functional program. Suppose we want to calculate a list with all numbers greater than 5 of a given list with numbers, e.g. `filter (>5) [1,7,2,6]` . In Miranda the definition of `filter` reads as follows, where `[]` denotes the empty list, and `(x:xs)` denotes an item `x` at the head of a list of items `xs`.

```
filter p []     = []
filter p (x:xs) = x : filter p xs , p x
                = filter p xs , otherwise
```

In this example, the Halstead metric values are $\eta_1 = 7$, $N_1 = 14$, $\eta_2 = 3$ and $N_2 = 11$. These values have been derived from the following < operator , frequency > tuples: < "( )" , 1 >, < "[ ]" , 2 >, < "," , 1 >, < "otherwise" , 1 >, < ":" , 2 >, < "=" , 3 >, < "filter" , 4 >. The < operand , frequency > tuples are: < "x ", 3 >, < "xs" , 3 >, < "p" , 5 >. We assume $\eta_2^* = \eta_2 = 3$ . The other Halstead quantities can be derived from these basic values.

The calculation of the McCabe metric value proceeds as follows. The number of arguments that are not identifiers is 2, resulting in a pattern complexity of 2. The number of guards is 1. The cyclomatic complexity number is 1 + (1 + 2) = 4.

### 3.2.3 Automated measurement

The metrics of Halstead and McCabe are based on the lexical and syntactical analysis of the program code. It is possible to use standard tools like a scanner and parser to automate the measurement of these metric values. In our study we used the Cornell Synthesizer Generator (Reps & Teitelbaum, 1989; Robbers, 1990). A schematic view of the analyser is given in Figure 3.2. (A similar metric analyser has been developed by Henry & Goff, 1989.)

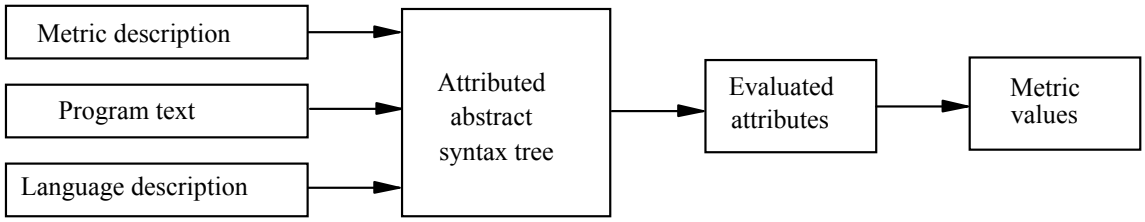



*Figure 3.2 Schematic view of the software metric analyser*

From a syntactical correct program text an abstract syntax tree is derived using the parse rules and a language description in the Extended Backus Naur Form. The metric values are calculated by means of attribute rules, provided in the metric description. For each node of the tree we can determine the complexity from the lower parts in the tree. The attribute rules have been developed for Pascal and Miranda in order to calculate the Halstead and McCabe metrics, and can be extended to other metrics. The output of the generator can be used in an auxiliary program to calculate the actual metric values.

## 3.3 Case Study

In the case study we explored the application of the syntactic complexity metrics described in the previous sections to programs in the functional programming language Miranda, and we established the correlation between these metrics and the readability of the programs. We used programs written in the language Pascal for comparison. The programs were taken from two groups of first year undergraduate students in Computer Science, after a programming course of one term. The (first) experimental group in Miranda and the second (control) group in Pascal. The students were asked to extend a existing program in the respective language. We compared their modifications of this reference program. The reference program carried out the conversion of a string representing a Roman number, e.g. MCLVII, to the corresponding decimal number, 1157. The extension of the program should allow the usual abbreviations in the input string, such as XL instead of XXXX. A conversion of the string MCXLVII should result in the decimal value 1147.

The readability of programs was assessed by 11 experts, all lecturers in Computer Science. They established a rank-order of readability for 9 modifications of the reference program in Pascal from the control group, and 8 modifications in Miranda from the experimental group. The agreement between the rankings is given by Kendall's coefficient of concordance W (Lindeman, Merenda & Gold, 1980). The results are given in Table 3.1. From these rankings we calculated the average expert rank of the programs in each group.

| Language | Number of programs | Correlation | Significance |
|---|---|---|---|
| Pascal | 9 | W = .74 | $\chi^2(8) = 65$ * |
| Miranda | 8 | W = .50 | $\chi^2(7) = 39$ * |

*Notes.* The correlation W is Kendall's coefficient of concordance.
The significance is tested by means of the $\chi^2$-test. *p < .001.

*Table 3.1 Correlation between expert rankings of readability for Pascal and Miranda programs*

Using the metric analyser, we determined the values of the software metrics for the reference programs and the modified programs in both languages. The ratings for the modified programs on Halstead's effort E and McCabe's cyclomatic complexity number were converted to rankings, or an average rank in the case of equal ratings. The correlation between these rankings and the average expert rank was calculated with Spearman's rank-order correlation coefficient $r_S$. The values of these correlations are given in Table 3.2. We will evaluate the results of this case study in the following section.

| Complexity 1 | Complexity 2 | Correlation | Significance |
|---|---|---|---|
| Extended Pascal programs ( N = 9 ) | | | |
| Effort | Cycl Compl Number | $r_s = .61$ | t(7) = 2.01 *** |
| Av Expert Rank | Effort | $r_s = .90$ | t(7) = 5.46 **** |
| Av Expert Rank | Cycl Compl Number | $r_s = .58$ | t(7) = 1.88 *** |
| Extended Miranda programs ( N = 8 ) | | | |
| Effort | Cycl Compl Number | $r_s = .89$ | t(6) = 4.78 **** |
| Av Expert Rank | Effort | $r_s = .38$ | t(6) = 1.01 * |
| Av Expert Rank | Cycl Compl Number | $r_s = .56$ | t(6) = 1.66 ** |

*Notes.* The correlation $r_s$ is Spearman's rank-order correlation coefficient.
The significance is tested by means of the Student t-test. *p<.20 **p<.10 ***p<.05 ****p<.01

*Table 3.2 Correlation between rankings of complexities for Pascal and Miranda programs*

## 3.4 Discussion

The metrics of Halstead and McCabe for the static syntactic complexity of programs in the functional programming language Miranda have been defined and the measurement has been automated. In the case study we compared the application of these metrics to programs in the functional language Miranda and the imperative language Pascal. A ranking of the expert assessment on readability was taken as a measure of the psychological complexity. From the values of the correlation between these expert rankings, we conclude that there is fair agreement between experts about the ranking of programs in Pascal ($W = 0.74$), but the agreement is low for programs in Miranda ($W = 0.50$). This could mean that there is not as much an accepted standard on readability for Miranda programs as for Pascal programs.

The correlation between the effort rank and the rank on the cyclomatic complexity number for the programs in Pascal is $r_S = 0.61$, and for Miranda $r_S = 0.89$. Both correlations are significant. The high value for Miranda indicates a consistent measurement of the syntactic complexity metrics as developed in this study.

The correlation between the average expert rank and the cyclomatic complexity order for Pascal is $r_S = 0.58$, and for Miranda $r_S = 0.56$. The correlation between the average expert rank and the effort order for Pascal is $r_S = 0.90$, and for Miranda $r_S = 0.38$. Both correlations for Pascal are high and significant, which is in agreement with the literature. The two correlations for Miranda are not significant. Obviously, this could be caused by the small number of programs used in this study. There are two reasons which could explain the low correlations for Miranda. Firstly, the value on the coefficient of concordance, indicating the agreement between experts on readability, for Miranda programs is low (see above). The second reason can be found in a more general argument given by Halstead (1977). He pointed to the dual role of the level of implementation of a program with respect to the understandability. For an expert the understandability is proportional to the program level, whereas for a novice the understandability is inversely proportional to the program level. In other linguistic studies, it was concluded that experienced writers cannot reliably predict the readability to novices of text in natural languages (Baker *et al.*, 1988).

Further study of software metrics for functional programming languages is required. There is a need to differentiate between metric values for various operators. Especially the use of higher-order functions can result in concise programs, and the count of such an operator should have a different contribution to the complexity than a simple arithmetic operator. It is necessary to estab-

lish the psychological complexity of each elementary language construct and the complexity of the compositions of these constructs in programs. This could be carried out in an axiomatic approach as outlined by Fenton (1991), along with contributions from measurement theory to software measurement (Weyuker, 1988; Zuse, 1991).

The complexity of the program text can be analysed at different linguistic levels: at lexical level, i.e. the vocabulary used in the program, at syntactical level, i.e. the linguistic structures in the program, and at the level of the program text as a whole, including the composition (sequencing and nesting) of linguistic structures. In this study we used the parsing of the program text by the computer as a model for the parsing by the human reader. The analysis should be based on a psychologically plausible parsing model, which is part of a cognitive model for the comprehension process. The Halstead metrics and McCabe metrics focus on the lexical and syntactic levels, and may not represent the most adequate levels for measurement of program comprehensibility (Card & Glass, 1990). We also have to incorporate the text level of analysis, as we see in discussions on readability and linguistic complexity of natural language texts (Frazier, 1988). The processing of program text has to be formulated in terms of comprehension of chunks of code and programming plans (Davies, 1989; Green & Borning, 1990), and the problem-solving capacities of programmers (Curtis, 1979). Furthermore, if we want to compare the understandability and the expressive power of programming languages we should weigh up metrics for programs and metrics for languages (Sammet, 1981).

Before metrics can be used in teaching programming as feedback to students on their programs, further development of software metrics is necessary together with the development of criteria to assess the readability of programs written in a functional programming language.

## Acknowledgement

I would like to thank J. van Merriënboer and H. Koppelman for their comments on an earlier version of this chapter; M. Massink for setting up the experiment with two programming groups; R. Couweleers and D. de Rooij for the implementation of the metric analyser and the collection of data[7]. I thank G. Kempen for bringing to my attention recent work by Green and Borning.

[7] This research was carried out in 1990. Sadly, R. Couweleers passed away in 1991.

# Part B : Modelling

## Issues

Many problems with software metrics are due to ill-conceived and poorly articulated models that underlie the metrics (Shepperd & Ince, 1994). The attribute to be measured should be made explicit. In Chapter 4, the attribute is the 'structure' of imperative programs. The traditional flowgraph model has been modified to capture nesting on conditional expressions in statements. The proposed structure graphs can be decomposed in prime structures by sequencing and nesting operations, as with traditional control-flow graphs. Software structure metrics based on flowgraphs and decomposition trees can also be used for structure graphs. In a separate paper (not included in this thesis), the approach to software structure metrics has been generalised to arbitrary sets of decomposition operations for flowgraphs (van den Broek & van den Berg, 1995).

Most of the modelling of programs has been carried out in the domain of imperative programming. In Chapter 5, the modelling of programs by flowgraphs and callgraphs is extended to functional programs. The proposed control-flow model captures the operational semantics of Miranda function definitions. Both types of abstractions, callgraphs and flowgraphs, are independent of the programming paradigm. Software metrics based on these abstractions can be used to compare attributes of programs. The modelling of Miranda type expressions by parse trees is described in part C (Chapters 6 and 7). Modelling as part of the metric development process is described in Chapter 8.

## Tools

Based on the flowgraph model and the callgraph model, a tool has been developed for the static analysis of Miranda programs. The tool is described in Chapter 5. The implementation is based on attributed grammars, as with the tool in Chapter 3 for the Halstead and McCabe metrics. This type of analysis is

supplementary to other static analyses performed on functional programs, such as strictness analysis and the type inference system (Peyton Jones, 1987; Plasmeijer & van Eekelen, 1993).

The use of such tools is quite diverse: for example during the software development or in the maintenance phase (e.g. Bache & Bazzana, 1994). They are used in reverse engineering and in anomaly checking. In an educational setting, these tools can be used to give students feedback on their software development process to compare their design and implementation. They have been used in assessment of the quality of students' programs (e.g. Ceilidh: Benford *et al.*, 1994).

The modelling of software is an essential stage in establishing assessment criteria for the quality of software. The development of tools is important to assure an objective assessment. However, once this has been achieved, one has to answer the question how valid the numbers are, and how these numbers can be used for different purposes. In fact, the Goal-Question-Metric paradigm (Basili & Rombach, 1988) reverses the order of these activities. In part C of this thesis, the issue of validation is addressed, with a case study on Miranda type expressions and with an experiment based on the flowgraph model for Miranda function definitions.

# Chapter 4

## 4. Modelling Software for Structure Metrics [8]

*In the traditional approach to structure software metrics, software is modelled by means of flowgraphs. A tacit assumption in this approach is that the structure of a program is reflected by the structure of the flowgraph. When only the flow of control between commands is considered this assumption is valid; it is no longer valid however when also the control flow inside expressions is considered. In this chapter, we introduce structure graphs for the modelling of software. Structure graphs can, just as flowgraphs, be uniquely decomposed into a hierarchy of indecomposable prime structures. We show how programs in an imperative language can be modelled by means of structure graphs in such a way that the structure of a program is always reflected by the structure of the corresponding structure graph.*

### 4.1 Introduction

In the traditional approach to structure software metrics (Fenton, 1991; Fenton & Kaposi, 1987), a program in an imperative language is mapped onto an abstract program, whereby program parts without structure are replaced by atomic actions; the resulting abstract program is mapped onto a flowgraph, and this flowgraph is decomposed into a hierarchy of primes (i.e. irreducible flowgraphs), which results in a decomposition tree onto which metric functions are applied. Where these metric functions are defined inductively, the metrics are called structure metrics. Flowgraphs will be defined formally in section 4.2 of this chapter. First, we will consider the modelling of imperative program fragments by flowgraphs.

An atomic action in a program is modelled by a $P_1$-flowgraph, which consists of a start node, a stop node, and one edge between these nodes. The flow-

[8] This chapter is an adaptation of: P.M. van den Broek & K.G. van den Berg (1993). Modelling Software for Structure Metrics. *Memoranda Informatica 93-12.* Enschede: University of Twente.

graph for an abstract program is constructed by associating a node with each atomic action, adding a stop node, identifying the node for the first atomic action as the start node, and drawing arcs from each node to its possible successors. Here the stop node is a successor for all possible last atomic actions. For instance, consider the following abstract program fragment:

```
WHILE a DO b END
```

This program fragment is modelled by the flowgraph $D_2$ in Figure 4.1.

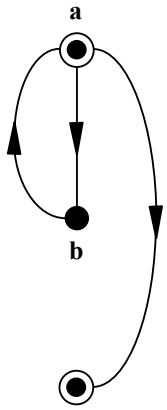



*Figure 4.1 Flowgraph $D_2$ of WHILE a DO b END*

Note that the node corresponding to `b` is a procedure node (has outdegree 1) and that the node corresponding to `a` is not a procedure node. This means that on node `b` another flowgraph can be nested, but on node `a` this is not possible. Consider the following program fragment:

```
IF c THEN d ELSE e END
```

This program fragment is modelled by the flowgraph $D_1$ in Figure 4.2.

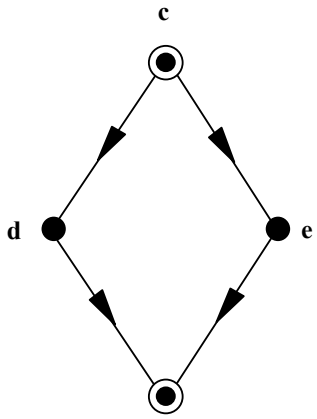



*Figure 4.2 Flowgraph $D_1$ of IF c THEN d ELSE e END*

Suppose we want to replace `b` in the program fragment `WHILE a DO b END` by `IF c THEN d ELSE e END`, resulting in the program fragment:

```
WHILE a DO
  IF c THEN d ELSE e END
END
```

Then, nesting the flowgraph $D_1$ (Figure 4.2) on node `b` of flowgraph $D_2$ (Figure 4.1) gives the flowgraph in Figure 4.3, denoted as $D_2(D_1)$.

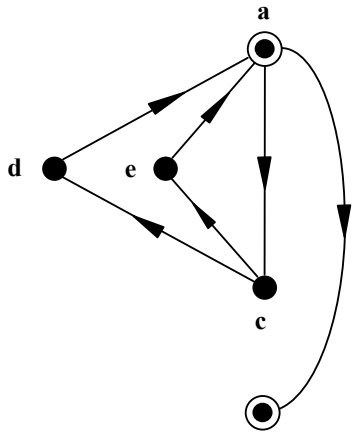



*Figure 4.3 Flowgraph of WHILE a DO IF c THEN d ELSE e END END*

Suppose we want to replace `a` in `WHILE a DO b END` by `c OR d`. Assuming a 'lazy' `OR`, the construct `c OR d` is modelled by the flowgraph $D_0$ which is given in Figure 4.4.

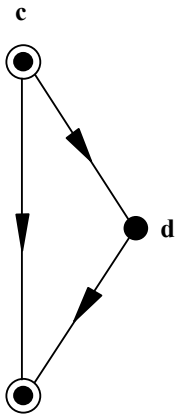



*Figure 4.4 Flowgraph $D_0$ of c OR d*

Now it is not possible to obtain the flowgraph of

```
WHILE c OR d DO b END
```

by nesting the flowgraph $D_0$ in Figure 4.4 on node a of the flowgraph $D_2$ in Figure 4.1. Instead, the flowgraph for this program, which is given in Figure 4.5, is a prime flowgraph to be called $X_1$.

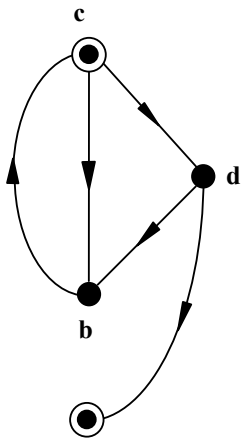



*Figure 4.5 Flowgraph $X_1$ of WHILE c OR d DO b END*

So, in this case the structure of the abstract program is not reflected by the structure of the corresponding flowgraph. In order to solve this problem, we will define the mapping from programs onto structure graphs, which are flowgraphs whose start node has outdegree 1.

This chapter is organised as follows. In section 4.2, we will recapitulate the theory of flowgraphs and flowgraph decomposition. In section 4.3 we will explain the notion of structure graph. Structure graphs can, analogous to flowgraphs, be uniquely decomposed into a hierarchy of prime structure graphs. Programs in an imperative language can be modelled by means of structure graphs in such a way that the structure of a program is always reflected by the structure of the corresponding structure graph. Structure metrics for these graphs are discussed in section 4.4. In the last section we consider two small example languages; we show how programs in these languages are mapped onto structure graphs.[9]

[9] This mapping has been implemented, and also the decomposition algorithm for the structure graphs, in the functional programming language Miranda.

## 4.2 Flowgraphs

In this section, we briefly recapitulate the theory of flowgraphs and their decomposition (Fenton, 1991; Fenton & Kaposi, 1987). We start with the definition of a flowgraph:

**Definition** A *flowgraph* is a 3-tuple (G,a,z) where G is a directed graph, and a and z are nodes of G, called *start node* and *stop node* respectively, such that:
- For each node x of G there is a path in G from a to z via x.
- The outdegree of z is 0.

The next two definitions specify operations on flowgraphs:

**Definition** If $F_1=(G_1,a_1,z_1)$ and $F_2=(G_2,a_2,z_2)$ are flowgraphs then the *sequence* $F_1;F_2$ of $F_1$ and $F_2$ is the flowgraph $(G_1;G_2,a_1,z_2)$ where $G_1;G_2$ is the directed graph which is obtained from the union of $G_1$ and $G_2$ by identifying the nodes $z_1$ and $a_2$.

**Definition** If $F_1=(G_1,a_1,z_1)$ and $F_2=(G_2,a_2,z_2)$ are flowgraphs and x is a node of $G_1$ with outdegree 1 (called a *procedure node*) then the *nesting* $F_1(F_2 \text{ on } x)$ is the flowgraph $(G_1(G_2 \text{ on } x),a_1,z_1)$ where $G_1(G_2 \text{ on } x)$ is the directed graph which is obtained from the union of $G_1$ and $G_2$ by deleting the edge whose source is x, identifying x and $a_2$, and identifying $z_2$ and the successor of x.

**Definition** The flowgraph $F_2=(G_2,a_2,z_2)$ is a *subflowgraph* of the flowgraph $F_1=(G_1,a_1,z_1)$ if $G_2$ is a subgraph of $G_1$ and $z_2$ is the source of all edges from $G_2$ to $G_1 \setminus G_2$.

**Definition** The subflowgraph $F_2=(G_2,a_2,z_2)$ of the flowgraph $F_1=(G_1,a_1,z_1)$ is a *one-entry subflowgraph* if
- the target of each edge from $(G_1 \setminus G_2) \cup \{z_2\}$ to $G_2$ is either $a_2$ or $z_2$, and
- if $a_1$ belongs to $G_2$ then $a_1=a_2$ or $a_1=z_2$

**Definition** The subflowgraph $F_2=(G_2,a_2,z_2)$ of the flowgraph $F_1=(G_1,a_1,z_1)$ is a *proper subflowgraph* if $G_1 \neq G_2$ and $F_2$ is not one of the two trivial flowgraphs. Here, the two *trivial flowgraphs* are the flowgraph consisting of one node only ($P_0$), and the flowgraph consisting of two nodes and one edge ($P_1$).

**Definition** The proper one-entry subflowgraph $F_2$ of the flowgraph $F_1$ is a *maximal one-entry subflowgraph* of $F_1$ if there exists no proper one-entry subflowgraph $F_3$ of $F_1$ such that $F_2$ is a proper one-entry subflowgraph of $F_3$.

The next two theorems show a way in which each flowgraph can be decomposed uniquely into a hierarchy of indecomposable flowgraphs (*primes*).

**Theorem** Each flowgraph F can be written uniquely as a *sequence* of nonsequential flowgraphs $F_1;F_2;..;F_n$.

**Theorem** Each nonsequential flowgraph F can be written uniquely as a simultaneous *nesting* $F_0(F_1 \text{ on } x_1,F_2 \text{ on } x_2,...,F_n \text{ on } x_n)$, where $F_1,F_2,...,F_n$ are the maximal proper one-entry subflowgraphs of $F_0$. Moreover, $F_0$ is a prime.

The nesting - in the last theorem - is usually denoted as $F_0(F_1, F_2,...,F_n)$, in which is abstracted from the nodes onto which the flowgraphs are nested.

An algorithm for the decomposition of flowgraphs is given in Bache & Wilson (1988). Note that, according to our definition of one-entry subflowgraphs, a requirement for $(G_2,a_2,z_2)$ to be a one-entry subflowgraph of $(G_1,a_1,z_1)$ is the absence of edges in $G_1$ from $z_2$ to nodes in $G_2$ other than $a_2$. This requirement is absent in the definitions in Fenton (1991), Fenton & Kaposi (1987) and Bache & Wilson (1988). Without this requirement, testing for the one-entry property in the algorithms above is not sufficient.

## 4.3 Structure graphs

As shown in the introduction, a drawback of the traditional way of modelling software by means of flowgraphs is that no refinement of conditions can be modelled, since conditions do not correspond to procedure nodes, and the operation of nesting is defined for procedure nodes only. The solution of this problem will lead to the introduction of a subset of flowgraphs, called structure graphs, as will be explained below.

As a first step towards a solution of this problem, we propose to model software by flowgraphs in such a way that each atomic action corresponds to a procedure node. For instance, `IF c THEN d ELSE e END` is modelled by the flowgraph of Figure 4.6, to be called structure graph $D_1'$ .

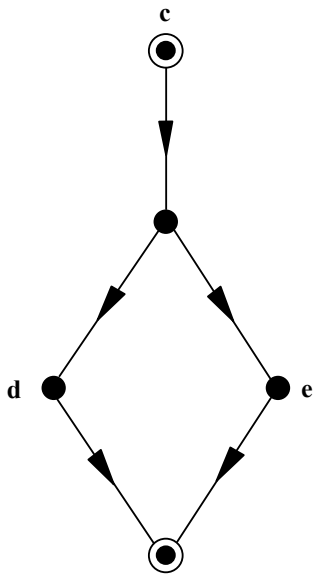



*Figure 4.6 Structure graph $D_1'$ of IF c THEN d ELSE e END*

Similarly, `WHILE a DO b END` is modelled by the flowgraph of Figure 4.7, to be called structure graph $D_2'$ .

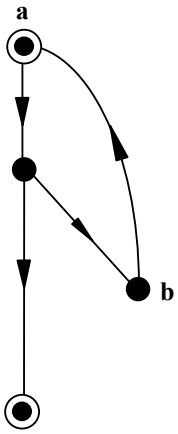



*Figure 4.7 Structure graph $D_2'$ of WHILE a DO b END*

Note that there are nodes with outdegree > 1 which do not correspond to atomic actions. After Whitty (1988), we call these nodes *select* nodes.

For each program, such a new representation as a flowgraph can be obtained by the old one by replacing in the former each node with outdegree > 1 by a procedure node with an edge to a new selection node. Unfortunately, this does not solve our problem. The program `WHILE c OR d DO b END` would correspond to the graph of Figure 4.8:

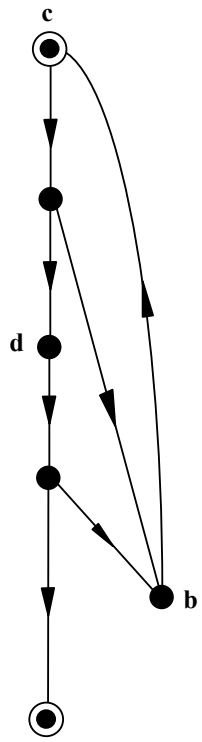



*Figure 4.8 Tentative structure graph of WHILE c OR d DO b END*

This is not what we want. Let us consider the flowgraph for `c OR d` in our new model, to be called structure graph $D_0'$ ,which is given in Figure 4.9:

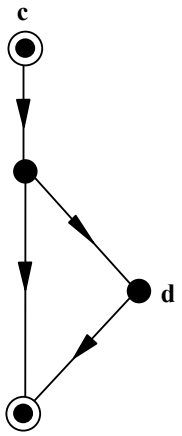



*Figure 4.9 Structure graph $D_0'$ of c OR d*

We want the structure graph for `WHILE c OR d DO b END` to be the structure graph $D_0'$ for `c OR d` (Figure 4.9) nested on node `a` of the structure graph $D_2'$ for `WHILE a DO b END` (Figure 4.7). This structure graph is shown in Figure 4.10.

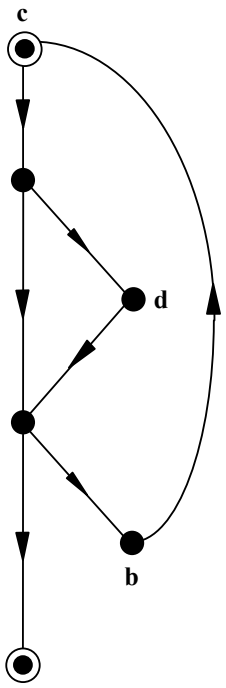



*Figure 4.10 Structure graph of WHILE c OR d DO b END*

In general, as the second step to the solution of our problem, we propose to assign graphs which can be interpreted as control flowgraphs only to 'basic' programs, and to assign to other programs graphs which are obtained from the graphs of their subprograms, using sequencing and nesting. In section 4.5 we illustrate this for two example languages.

We are left with one major problem. The graphs assigned to 'basic' programs should be primes. However, the graph for `IF a THEN b ELSE c END`, which is given in Figure 4.6, is not a prime; it is a sequence $P_1;D_1$ of the prime flowgraphs $P_1$ and $D_1$. The same is true for the graph in Figure 4.9, which corresponds to the basic program `c OR d` , which is the sequence $P_1;D_0$. The third (and final) step to the solution of our problem is therefore to consider only a subset of the flowgraphs, called structure graphs. A *structure graph* is a flowgraph whose start node is a procedure node.

This choice is justified by the observation that programs should start with an action, not with a selection. Note that the flowgraphs in Figure 4.6 and Figure 4.9 are structure graphs which cannot be decomposed into smaller structure graphs, i.e. they are prime structure graphs, respectively $D_1'$ and $D_0'$.

The theory of structure graphs is, fortunately, analogous to the theory of flowgraphs. The operations of sequencing and nesting are well-defined on structure graphs, and structure graphs can be decomposed uniquely into a hi-

erarchy of prime structure graphs. An algorithm for the decomposition of structure graphs may be obtained from an algorithm for the decomposition of flowgraphs in a straightforward way. However, the result of the decomposition of a structure graph into prime structure graphs can be quite different from the result of the decomposition of the same graph into the traditional prime flowgraphs. Consider for instance the graph in Figure 4.11.

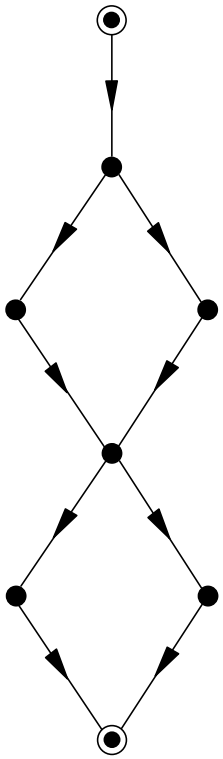


*Figure 4.11 Example graph*

As a traditional flowgraph, its decomposition is a sequence of three prime flowgraphs: P1; $D_1$'; $D_1$'. As a structure graph, its decomposition is a nesting of the prime structure graph $D_1$' on the prime structure graph $D_1$': i.e., $D_1'(D_1')$.

## 4.4 Structure metrics

The prime decomposition of flowgraphs has been used for the definition of the important class of structure metrics (Fenton, 1991). These metrics can be described completely in terms of the primes and the operations of sequencing and nesting. A *structure metric* m is determined uniquely by the following three characteristics:

1. m(F) for each prime F
2. a function $g_n$ such that $m(F_1;...;F_n) = g_n(m(F_1),...,m(F_n))$
3. a function $h_F$ such that $m(F(F_1,...,F_n)) = h_F(m(F_1),...,m(F_n))$ for each prime F.

A structure metric with these properties is called a hierarchical metric. Moreover, if the nesting function *h* is independent of F then the metric is called a recursive metric. So, the class of recursive metrics is contained in the class of hierarchical metrics.

For example, the structure metric *depth of nesting* $m_d$ is defined as follows (Fenton, 1991):

1. $m_d(P_1) = 0$
   for each prime $F \neq P_1$: $m_d(F) = 1$
2. $m_d(F_1;...;F_n) = \max(m_d(F_1),...,m_d(F_n))$
3. $m_d(F_0(F_1,...,F_n)) = 1 + \max(m_d(F_1),...,m_d(F_n))$

For structure graphs, structure metrics can be defined in the same way. It should be kept in mind that the decomposition for structure graphs, differs from flowgraphs as used in the structure metrics given above, as discussed in the previous section. The metric values need not be the same in both approaches. E.g., the depth of nesting for `WHILE c OR d DO b END` in the traditional modelling (see Figure 4.5) is 1 and in the new model (see Figure 4.10) the depth of nesting is 2, i.e. the depth of $D_2'(D_0')$.

## 4.5 Two small languages

In this section we consider the mapping from programs of some small languages to structure graphs. We do not consider the mapping from programs to abstract programs; our languages themselves consist of abstract programs. Our first language is given in Table 4.1:

```
<program>     = PROGRAM <name> ; BEGIN <body> END <name> .
<body>        = <expression> | <expression> ; <body>
<expression>  = S | WHILE <expression> DO <body> END |
                IF <expression> THEN <body> ELSE <body> END
<name>        = <letter> | <letter> <name>
<letter>      = 'a' | ..... | 'z'
```

*Table 4.1 Sample programming language*

A program consists of a body, which equals a sequence of expressions. There are three kinds of expressions: the `WHILE` expression, the `IF` expression, and the expression `S`. This last expression corresponds to atomic actions.

The mapping from programs to structure graphs is defined by induction. The structure graph of a program is the structure graph of its body. The structure graph of a body is the sequence of the structure graphs of its expressions.

The structure graph of the expression S is the structure graph with two nodes and one edge. The structure graphs of a IF expression and an WHILE expression are the structure graphs of Figure 4.6 and Figure 4.7 respectively, on which the structure graphs of their subexpressions are properly nested.

It is easily shown that each expression is mapped onto a nonsequential structure graph. From this it follows that the decomposition tree of the structure graph of each program can be obtained from the parse tree of the program, and vice versa. So, for this language there is no need to construct and decompose a structure graph in order to obtain the structure of a program; the structure is completely determined by the syntax of the program. This remains true when we add more 'structured' expressions, like REPEAT-loops, and OR and AND expressions.

Our second example language is obtained from the first one by adding labelled expressions and a GOTO expression (see Table 4.2):

```
<program>     = PROGRAM <name> ; BEGIN <body> END <name> .
<body>        = <expression> | <expression> ; <body> |
                <label> : <expression> |
                <label> : <expression> ; <body>
<expression>  = S | WHILE <expression> DO <body> END |
                IF <expression> THEN <body> ELSE <body> END |
                GOTO <label>
<name>        = <letter> | <letter> <name>
<label>       = <digit>  | <digit>  <label>
<letter>      = 'a' | ..... | 'z'
<digit>       = '0' | ..... | '9'
```

*Table 4.2 Extended sample programming language*

Since we will not assign a structure graph to a GOTO expression, the mapping from programs to structure graphs cannot be defined by induction in this case. Informally, the structure graph corresponding to a program in this language is obtained by first replacing the GOTO expressions by S and constructing the structure graph as in the first example language (forgetting about the labels) and then removing the GOTO-nodes by redirecting the incoming arcs for each GOTO-node to the start node of the structure graph corresponding to the expression with the appropriate label. More formally we proceed as follows.

**Definition** A *generalised structure graph* is a 7-tuple consisting of
- a set N, the elements of which are called nodes,
- a node a, called the start node,
- a node z, called the stop node,

- a set E of ordered pairs of nodes, the elements of which are called edges,
- a set L, the elements of which are called labels,
- a set B of ordered pairs of a label and a node, the elements of which are called bindings,
- a set D of ordered pairs of a node and a label, the elements of which are called dangling edges

A structure graph can be seen as a generalised structure graph for which the sets L, B and D are empty. Sequencing and nesting are defined for generalised structure graphs just as for structure graphs (if dangling edges are treated as 'real' edges).

A mapping from programs to generalised structure graphs can be defined by induction as follows.

The generalised structure graph of a program is the generalised structure graph of its body. The generalised structure graph of a body is the sequence of the generalised structure graphs of its (labelled) expressions. The generalised structure graph of a labelled expression is the generalised structure graph of the expression to which a binding is added consisting of the label and the start node. The generalised structure graph of the expression `S`, of a `WHILE` expression and of an `IF` expression are the structure graphs as in the previous example, on which the generalised structure graphs of their subexpressions are properly nested. Finally, the generalised structure graph of a `GOTO` expression consists of two nodes, the start node and the stop node, and a dangling edge consisting of the start node and the label.

Having obtained a generalised structure graph for a program, its dangling edges can be replaced by 'real' edges; their target nodes can be obtained from the set of bindings. If no binding for a label is found, the program is not well-formed. If for all labels bindings are found, either the result is a structure graph or the program contains unreachable code. So we have defined a mapping from well-formed programs to structure graphs, where the well-formed programs are programs without missing labels and without unreachable code. The procedure nodes of the structure graph correspond to basic actions and to `GOTO` statements. The nodes corresponding to `GOTO` statements are indirection nodes, and can be removed. Note that the syntax of our language allows a `GOTO` expression as the conditional of a `WHILE` or an `IF` expression, but that this is impossible for a well-formed program.

As an example, consider the rather unusual[10] program given in Table 4.3:

[10] For example in Pascal, such a program is not allowed according to the ISO standard

```
PROGRAM example;
BEGIN
   IF S1 THEN GOTO 1 ELSE S2 END;
   IF S3 THEN 1:S4 ELSE S5 END
END example.
```

*Table 4.3 Example program*

The structure graph and the (traditional) flowgraph of this program is given in Figure 4.12.

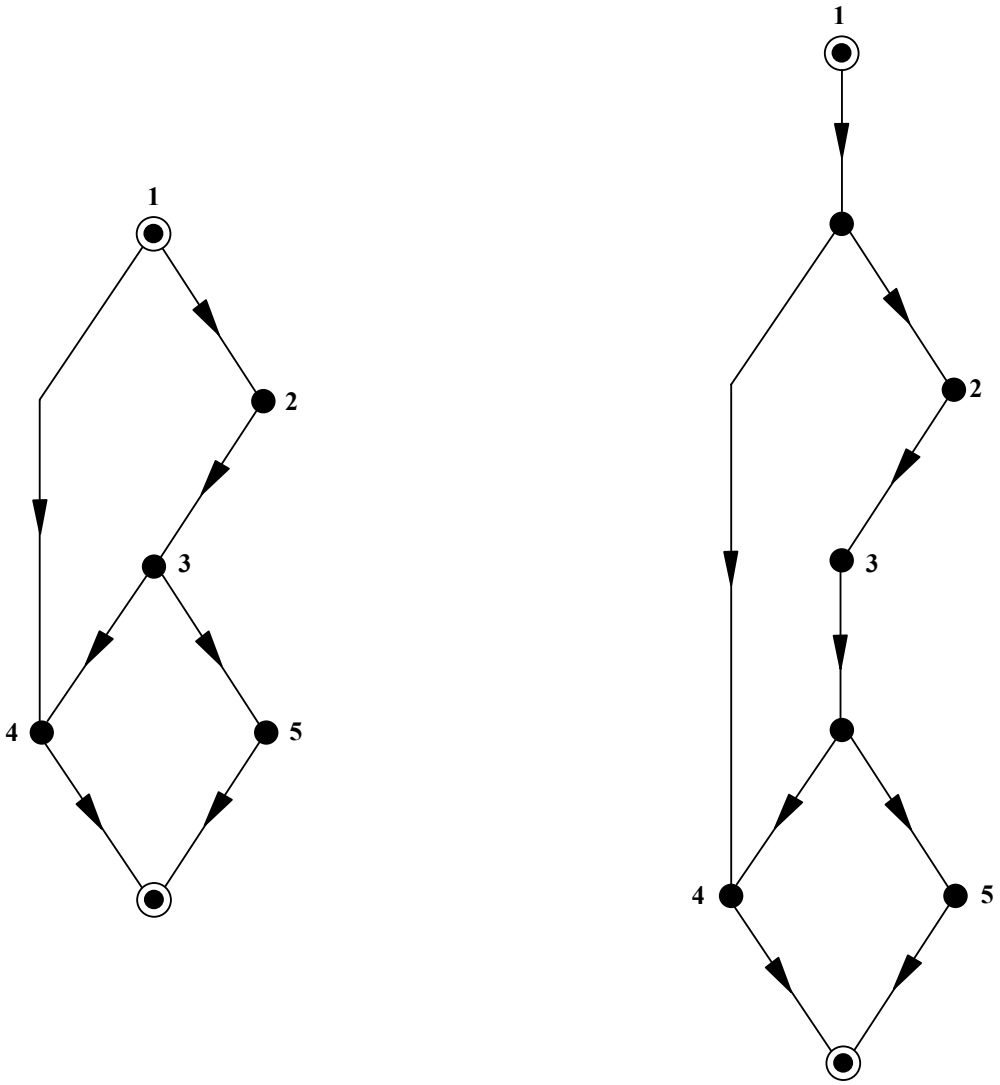



*Figure 4.12 Flowgraph (left) and structure graph (right) of the example program in Table 4.3*

The occurrences of `S` in the example program have been given indices; these indices are used in the graphs to show the correspondence between atomic actions and graph nodes. The flowgraph of Figure 4.12 is a prime flowgraph; the structure graph of Figure 4.12 however is not a prime structure graph: it contains as substructure the sequence of $S_2$ and $S_3$. This is an example where our structure graph approach reveals a substructure which remained unnoticed in the traditional flowgraph approach. It is also interesting to note that this sequential substructure was not explicitly denoted as a sequence in the program.

## 4.6 Conclusion

We have introduced the modelling of programs in terms of structure graphs, which are flowgraphs whose start node is a procedure node. Structure graphs can, just as flowgraphs, be uniquely decomposed into a hierarchy of indecomposable prime structures. It has been shown that structure graphs are better suited than flowgraphs to model the structure of programs in an imperative language. We have given explicitly the mapping from programs of small example languages to structure graphs.

# Chapter 5

## 5. Static Analysis of Functional Programs [11]

*In this chapter, the static analysis of programs in the functional programming language Miranda is described based on two graph models. A new control-flow graph model of Miranda definitions is presented, and a model with four classes of callgraphs. Standard software metrics are applicable to these models. A Miranda front end for Prometrix[12], a tool for the automated analysis of flowgraphs and callgraphs, has been developed. This front end produces the flowgraph and callgraph representations of Miranda programs. Some features of the metric analyser are illustrated with an example program. The tool provides a promising access to standard metrics on functional programs.*

### 5.1 Introduction

Static analysis of programs has the potential to contribute to the control of quality of software. Internal attributes, such as structural properties, measured in the static analysis, are claimed to have a correlation with external attributes, such as comprehensibility, maintainability and testability. Traditionally, static analysis and related tools focuses mainly on programs written in imperative programming languages (Fenton, 1991). In this chapter, two models for static analysis, control-flow graphs and callgraphs, will be elaborated for the analysis of programs written in the functional programming language Miranda (Turner, 1986) with respect to the comprehensibility of programs (Davies, 1993). The measurement and validation of internal attributes on size and structure based on these models are addressed. The validation of the models with respect to external attributes are subject of a separate study (van den Berg & van den Broek, 1995b; see Chapter 9 of this thesis).

---

[11] K.G.van den Berg & P.M. van den Broek (1995). Static Analysis of Functional Programs, *Information and Software Technology, 37(4),* 213-224.

[12] Prometrix is a product of Infometrix Software

Callgraphs are used to model dependencies between program constructs, such as functions or modules. Callgraphs are related with hierarchy charts as used in several structured design methods (Yourdon & Constantine, 1979). They capture the dependencies of objects in the program at different levels of abstraction. E.g., one may define a callgraph for dependencies between functions within a module; or dependencies between modules, and so on. The root node of the callgraph corresponds to the highest level object. Callgraphs are used in static program analysers (Bache, 1990). Callgraphs for Prolog programs have been given by Fenton & Kaposi (1989). A callgraph model for functional programs in Miranda has been described by Harrison (1993). In this chapter, four classes of callgraphs will be introduced.

There are different aspects of control-flow in functional programming. One important aspect is determined by the reduction strategy for the evaluation of expressions. In Miranda, the functional programming language studied here, this strategy is normal order reduction, also called lazy evaluation (Bird & Wadler, 1988). Another aspect of control-flow is related to the syntactical structure of the function definitions in programs. This aspect, that usually gets little attention, will be addressed in this chapter.

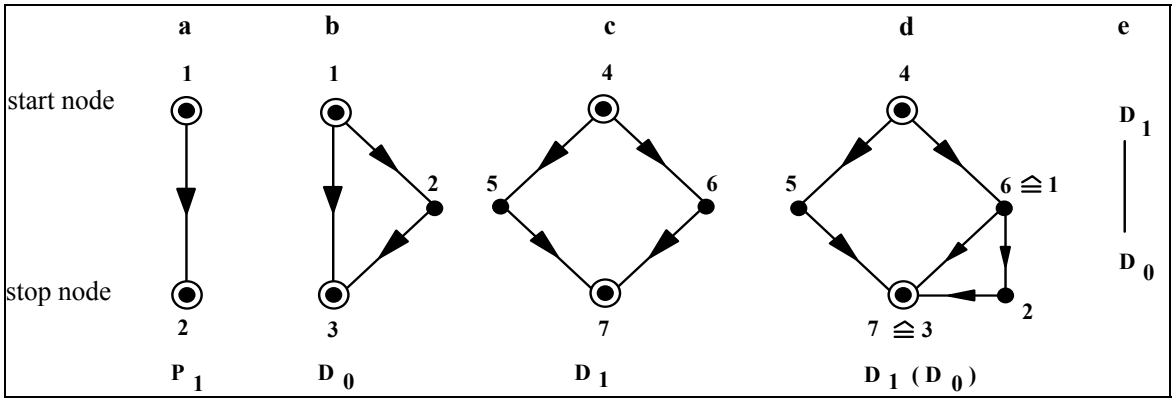



*Figure 5.1 Elementary flowgraphs and decomposition tree*

Flowgraphs are used for the modelling of control-flow in imperative programs (Fenton, 1991). The nodes in the directed graphs correspond to statements in the programs, whereas the edges from one node to the other indicate a flow of control between corresponding statements. The stop node in a flowgraph has outdegree zero, and every node lies on some path from the start node to the stop node. The nodes with outdegree equal to 1 are called procedure nodes; all other nodes are termed predicate nodes. E.g., an elementary action is modelled as flowgraph in Figure 5.1a (referred to as $P_1$); the if-then construct in a program is modelled as flowgraph in Figure 5.1b (referred to as $D_0$); the if-then-else construct is modelled as flowgraph in Figure 5.1c (referred to as $D_1$).

Flowgraphs can be concatenated (sequencing) to a new flowgraph; and flowgraphs can be nested on another. An example of nesting $D_0$ onto $D_1$ at node 6 in Figure 5.1c, is given in Figure 5.1d. This is denoted as $D_1(D_0)$, in which is abstracted from the node onto which is nested. Associated with any flowgraph is a decomposition tree which describes how the flowgraph is built by sequencing and nesting elementary flowgraphs, such as $D_0$ and $D_1$. The decomposition tree of the flowgraph in Figure 5.1d is depicted in Figure 5.1e.

In order to quantify internal attributes of software, metrics have been defined on flowgraphs, decomposition trees and callgraphs (Fenton, 1991). These metrics can be divided into two main classes: size metrics (e.g. number of nodes and edges) and structure metrics (e.g. nesting depth and width, based on a decomposition in primitive components). Several of the standard metrics will be used on the models discussed in this chapter.

This chapter is organised as follows. First, more details about programs in the functional programming language Miranda will be given by explaining an example program. Furthermore, the modelling of the control-flow and dependencies in the callgraph for functional programs will be elaborated on. The actual data of some software metrics for the example program will be described. The final sections discuss the Miranda analyser and some results obtained with this approach.

## 5.2 Functional programs

In this section, some characteristics of programs in the functional language Miranda (Turner, 1986; Bird & Wadler, 1988) will be described with an example program.

### 5.2.1 Example program

In Table 5.1, an example program, usually called a script, is given. The line numbers are added for further explanation.
The function `main` (lines 4-7) returns the sum of the `j`-th through `k`-th complex number in `list`, in which each complex number is derived from a list of (integer or real) numbers as follows: an empty list will give complex number `0 + 0 i`, a list with one number `x` will give complex number `x + 0 i`, and a list with two or more numbers `x,y,...` will give complex number `x + y i`. Informally, the function `main` can be specified as follows:

$$\texttt{main j k } [c_1,\ldots,c_j,\ldots,c_k,\ldots,c_n] = c_j + \ldots + c_k$$

For the given test data (line 10) and with `j = 1` and `k = 4`, the (top) expression `main 1 4 test` evaluates to the string `"13 + 5 i"`.

```
|| file complex.m                                                    1
|| main j k list   is the sum of the j-th through k-th              2
|| complex number in list                                           3
main ::  num -> num -> [[num]] -> [char]                            4
main j k list                                                      5
  = showct (sumlist sublist)                                       6
    where sublist = take (k-j+1) (drop (j-1) list)                 7
                                                                     8
|| test data                                                        9
test =  [[4,5],[1,0],[8],[],[2,3,4],[7,8]]                        10
                                                                    11
|| specification complex numbers                                  12
|| re(rect(a,b)) = a                                              13
|| im(rect(a,b)) = b                                              14
                                                                    15
|| type definition complex numbers                                16
abstype ct                                                         17
with                                                               18
   rect   :: (num,num) -> ct                                       19
   re     :: ct -> num                                             20
   im     :: ct -> num                                             21
   showct :: ct -> [char]                                          22
                                                                    23
|| implementation complex numbers                                24
ct          == [num]                                               25
rect (a,b)  = [a,b]                                                26
re [a,b]    = a                                                    27
im [a,b]    = b                                                    28
showct z    = x, if im z = 0                                       29
            = y ++ " i", if re z = 0                               30
            = x  ++ " + " ++  y  ++ " i", otherwise                31
              where (x,y) = (shownum(re z), shownum(im z))         32
                                                                    33
|| derived operations complex numbers                            34
plus        :: ct -> ct -> ct                                      35
c1 $plus c2 = rect (re c1 + re c2, im c1 + im c2)                  36
                                                                    37
|| sum of complex numbers in list                                38
|| each complex number is derived from a list of numbers         39
sumlist              :: [[num]] -> ct                              40
sumlist []           = rect(0,0)                                   41
sumlist ([x1,x2]:xss)= c $plus sumlist xss                         42
                       where c = rect(x1,x2)                       43
sumlist (xs:xss)     = sumlist xss, if #xs = 0                     44
                     = c $plus sumlist xss, if #xs = 1             45
                     = sumlist ((take 2 xs):xss),otherwise         46
                        where c = rect(x,0)                        47
                                  where x = hd xs                  48
```

*Table 5.1 Example Miranda program*

For complex numbers, an abstract data type is given: the specification as comment (lines 12-14) and the type definition of the base operations (lines 17-22). Any text on a line after two vertical bars is comment (e.g. lines 1-3). In the implementation (lines 26-32) a complex number is represented by a list of numbers, given by the type synonym symbol == (line 25). The derived operation plus (line 36) is defined in infix notation (name of the function with a $-prefix). With the reserved word *where* the local definitions are indicated (e.g. line 7). On line 32, x and y are defined simultaneously in a so called compound definition. The other functions in this script (take, drop, shownum, hd, ++ and #) are Miranda library functions.

For each function the type of the function is provided: the name of the function followed by a double colon and a type expression (e.g. line 4). The right arrow → in the type expression denotes a function type.

The example program could have been programmed more proficiently, especially the function sumlist, and with a more distinct specification of the functions. However, this rather inexpert implementation will be used to exemplify several modelling issues.

### 5.2.2 Structure of function definitions

A script consists of a number of definitions. A definition consists of a number of clauses. A clause consists of a number of cases, possibly followed by a script with the local definitions of that clause. This structure will be illustrated with the function sumlist (see Table 5.2).

```
sumlist []              = rect(0,0)                                    41

sumlist ([x1,x2]:xss)   = c $plus sumlist xss                          42
                          where c = rect(x1,x2)                        43

sumlist (xs:xss)        = sumlist xss, if #xs = 0                      44
                        = c $plus sumlist xss, if #xs = 1              45
                        = sumlist ((take 2 xs) : xss), otherwise       46
                          where c = rect(x,0)                          47
                                where x = hd xs                        48
```

*Table 5.2 Structure of the definition sumlist*

The definition *sumlist* consists of three clauses (starting at line 41, 42 and 44). The first clause consists of one case (line 41). The second clause consists of one case (line 42), followed by a local script with the definition of c (line 43: single clause, single case). The third clause consists of three cases (lines 44-46), followed by a local script with the definition of c (line 47: single clause, single case with a local script with the definition of x at line 48).

## 5.3 Control-flow model

The control-flow, as reflected in the syntactic structure of the function definitions, is determined by the order of the clauses and the patterns, and the order of the cases and the guards. A detailed account on pattern-matching and guards in Miranda is given by Peyton Jones and Wadler (Peyton Jones, 1987). From other aspects of the control-flow in the actual evaluation of expressions, such as laziness (Bird & Wadler, 1988), will be abstracted from.

### 5.3.1 Control-flow in function definitions

The clauses are selected by matching the patterns in the arguments. For example, the first pattern in the function sumlist (see Table 5.1) is an empty list [ ] (line 41); the second pattern ([x1,x2]:xss) is a non-empty list with a head-element consisting of a list with two elements (line 42). Here, there is a pattern within another pattern. The pattern (xs:xss) in the third clause (line 44) is again a non-empty list, but more general than the pattern in the previous clause: any head-element will match. The pattern in the first clause will be checked first, then the second, and so on. Only if all patterns in the clauses are disjoint and exhaustive, the clauses can be written in any order. There are patterns which always match, e.g. the pattern z in the definition of showct (line 29). If no pattern succeeds there is an error in the definition.

If a clause is selected, the cases in a clause are selected by the guards of each case. There are no guards in the first and second clause. The first guard in the third clause (line 44) is the test (#xs=0), the second guard is (#xs=1), the last guard (line 46) is *'otherwise'* which will succeed always. The topmost guard will be checked first, then the second, and so on. E.g., in the second case of the function showct (line 30), it is assumed that the first guard resulted in the value False, so that in this case (im z ≠ 0). Only if all guards are disjoint and exhaustive, the cases can be written in any order. If no guard succeeds, which may happen if there is no *'otherwise'* guard, in Miranda the following

function clause will be checked [13]. If there is no other clause there will be a program error.

### 5.3.2 Modelling control-flow in function definitions

In the mapping of a program to a model, one has to keep in mind for which purpose the model will be used. A model for the testability of a program could be different from a model for the comprehensibility (Shepperd & Ince, 1993). In the subsequent modelling of the control-flow, internal attributes relevant to the external attribute comprehensibility of functional programs have to be captured. Eventually, this modelling has to be validated.

For the static analysis, arguments in a function clause with patterns that may fail will be modelled as one predicate node with outdegree 2. Patterns that never fail consists of just one or more distinct identifiers, e.g. the pattern z in the definition of showct (line 29). A pattern that always succeeds will not be modelled as a node in the flowgraph.

In Miranda, common used patterns in function definitions that may fail are:

- patterns with a constant: real, integer, character, string
- patterns with constructors: user defined algebraic constructors, or standard constructors for a list (line 27,28 and in line 41, 42 of the function sumlist)
- patterns with the + operator, e.g. n+1 where n is an integer
- patterns with the list-constructor **:** , such as in (xs:xss) in line 44
- multiple occurrences of variables: two or more times the same identifier in the patterns

Multiple patterns, such as in the second clause of sumlist (line 42) or patterns in two or more arguments, will be modelled just as one predicate node. Moreover, we will abstract from the actual content of patterns. E.g., the two patterns [ ] and (xs:xss) cover all possible list arguments (the function is total). However, both patterns will be modelled with a predicate node, as if they were independent.

Guards will be modelled as predicate nodes with outdegree 2. Again, we will abstract from the actual content of the guard. E.g., a guard with just the boolean value True, or the boolean expression (1=1), will be modelled as a predicate node. Composite guards are modelled just as one predicate node. The guard '*otherwise'* will not be modelled with a node in the flowgraph.

Expressions other than guards on the right hand side of the function definition will be modelled just as one procedure node. In the modelling, we will ab-

[13] In some implementations of functional languages, the program will not proceed with the following clause and a program error will be reported

stract from the actual content of these expressions, which be may very simple (line 27) or more complicated (line 7).

In this flowgraph modelling of functional programs, there is no recursion and there are no iterative constructs, such as the while-do structure in an imperative language. In terms of prime flowgraphs, there are no $D_2$ (while-do) and $D_3$ (repeat-until) structures. Furthermore, there is no sequencing of flowgraphs in this model.

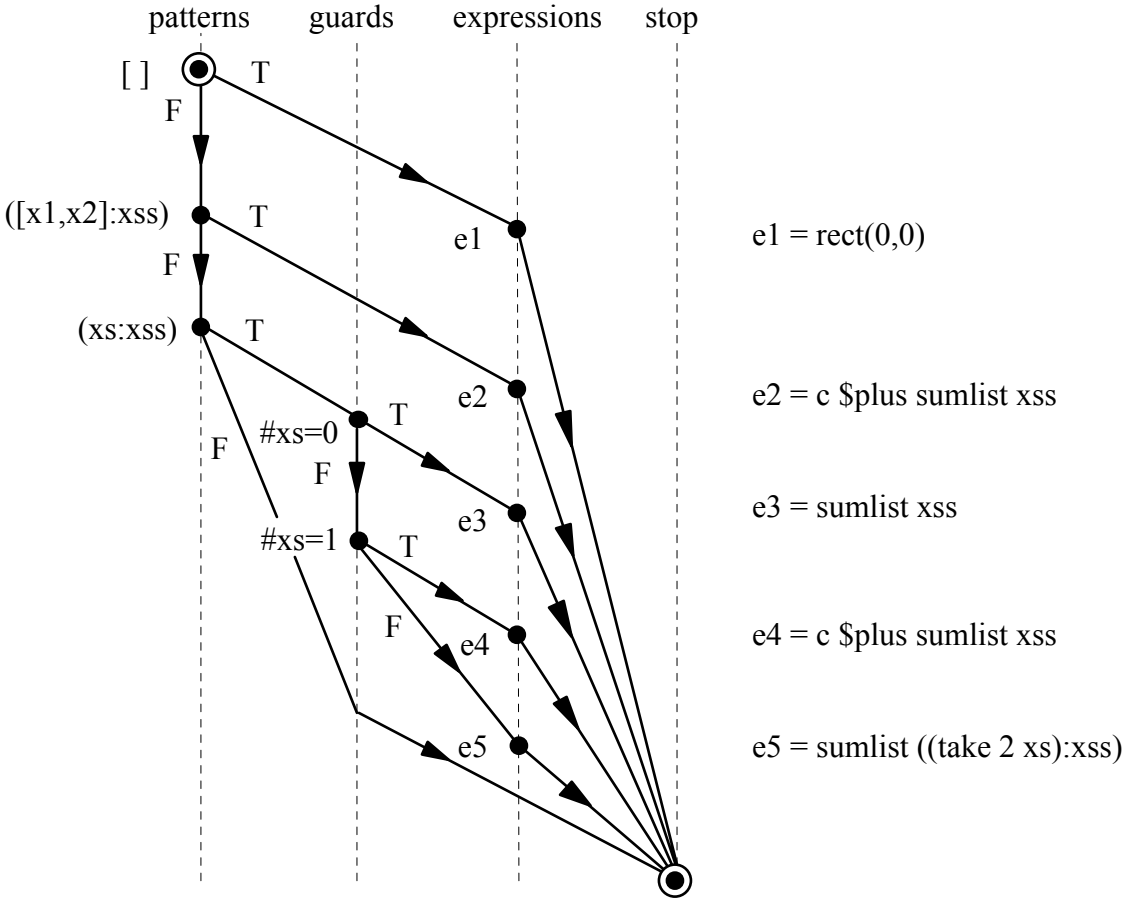



*Figure 5.2 Annotated control-flow graph of the function sumlist*

### 5.3.3 Control-flow graph and decomposition tree

From the modelling discussed in the previous section, the control-flow graph for the function sumlist is given in Figure 5.2. The four vertical lines indicate the kind of nodes in the flowgraph: predicate nodes (outdegree 2) for patterns and guards, procedure nodes (outdegree 1) for the expressions, and finally the stop node (outdegree 0). For the predicate nodes, the True (T) and False (F)

branches are indicated. Note that the lower (False) branch starting at the pattern (xs:xss) is infeasible because either the pattern [ ] or the pattern (xs:xss) will succeed: these two patterns are exhaustive. However, as described in the previous section, in this model will be abstracted from the actual content of the patterns, and the pattern (xs:xss) will be modelled as a predicate node with outdegree 2.

The decomposition tree of flowgraph can be derived by a hierarchical decomposition in prime flowgraphs (Fenton, 1991). The decomposition tree of the function sumlist is given by:

$D_1(D_1(D_0(D_1(D_1))))$

and can be depicted as a tree without branches (cf. Figure 5.1e).

There are simple function definitions resulting in flowgraphs that are not D-structured (i.e. containing other than $D_0$, $D_1$, and $P_1$-primes). Consider for example the following function f (the function funnyLastElt in Peyton Jones, 1987: p 58):

The function f returns the last element of its argument list, except that if a negative element is encountered then it is returned instead.

```
f (x:xs) = x, if x < 0                    (1)
f (x:[]) = x                              (2)
f (x:xs) = f xs                           (3)
```

The function f is a partial function, defined for non-empty lists only. The clause numbers are added. The annotated flowgraph of this function definition is given in Figure 5.3a.

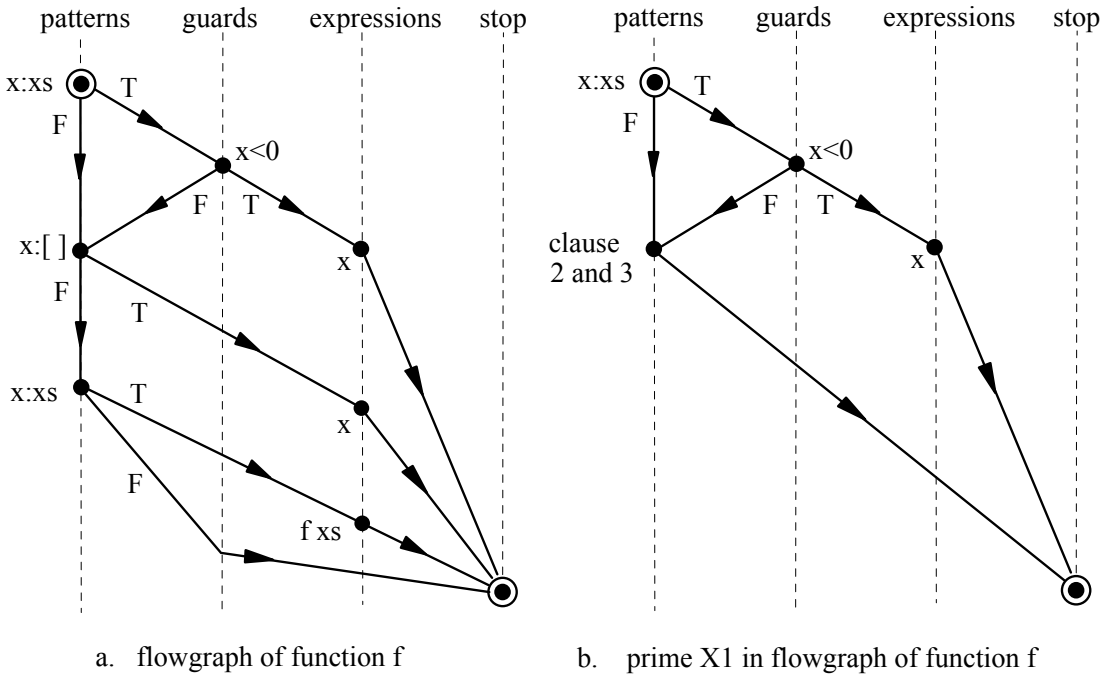
The decomposition of this flowgraph is $X_1(D_1(D_0))$, where $X_1$ is the prime given in Figure 5.3b. In imperative languages, this prime is associated with a lazy boolean AND-expression in a selection (cf. Fenton & Kaposi, 1987).

Furthermore, from this example it can be shown that guards interact with pattern matching and the order of the clauses. There are 6 permutations of the order of the three clauses in the function f. Only two of them, (1,2,3) and (2,1,3), give a definition which satisfies the specification.

An alternative definition of the function f with the same functionality is the following function f' :

```
f'(x:xs) = x    , if  x < 0  \/  xs = []
         = f' xs, otherwise
```

The flowgraph belonging to this function f ' is D-structured; its decomposition is $D_0(D_1)$. The composite guard, in this example consisting of a lazy boolean or-expression, is modelled as one predicate node, as has been described in the previous section.




a. flowgraph of function f

b. prime X1 in flowgraph of function f

*Figure 5.3 Annotated control-flow graph of the function f with prime $X_1$*

Whether this alternative definition, with a D-structured flowgraph decomposition, should be preferred, e.g. with respect to the external attribute comprehensibility, to the first definition with the X-prime in its flowgraph decomposition, has to be established in a separate validation study (van den Berg & van den Broek, 1995b; see Chapter 9 of this thesis).

### 5.3.4 Flowgraph metrics

There are a large number of metrics defined on flowgraphs and decomposition trees (Fenton, 1991). A selection of flowgraph metrics for the function sumlist is given in Table 5.3. A short description of the metrics will be given. The size metrics give the number of nodes and edges in the flowgraph. The local structure metrics give the occurrences and sizes of the primes in the decomposition. The overall structure metrics give some classical measures on flowgraphs: e.g. the cyclomatic complexity number of McCabe. Testability metrics can be computed from the decomposition tree provided that the values can be computed for the primes as well as for nesting and sequencing (Fenton, 1991). In tools,

like Qualms(1988) and Prometrix(1993), the prime decomposition is used in the computation of the testability metrics.

| Metric | | Value |
|---|---|---|
| Size Metrics | | |
| – | Number of nodes | 11 |
| – | Number of edges | 15 |
| Local Structure Metrics | | |
| – | Is D-structured | 1 |
| – | Occurrences of $D_0$ | 1 |
| – | Occurrences of $D_1$ | 4 |
| – | Occurrences of exotic primes | 0 |
| – | Biggest prime | 4 |
| – | Depth of nesting | 5 |
| Overall Structure Metrics | | |
| – | McCabe's metric | 6 |
| – | Prather's metric | 32 |
| – | Basili-Hutchens SynC | 12.21 |
| Testability Metrics | | |
| – | Statement testability | 5 |
| – | Branch testability | 6 |

*Table 5.3 Flowgraph metrics for the function sumlist*

In the modelling of functional programs, and the special situation with only $P_1$, $D_0$ (if-then) and $D_1$ (if-then-else) structures and no sequencing, the following testability metrics will give equal values: all-path testing, visit-each-loop path testing, simple path testing and branch testing. Therefore, only one of these metrics, branch testability, is included in the selected metrics of Table 5.3. If 'exotic' prime structures are encountered in the flowgraph, here primes other than $D_0$, $D_1$ and $P_1$, the testability metrics for these primes have to be added.

The testability metrics give the number of test cases required in each of the testing strategies. E.g., branch testing requires that each edge in the flowgraph be visited at least once; for the function sumlist a minimum of 6 test cases is required. Statement testing requires that each node in the flowgraph be visited at least once. The test cases can directly be derived from the flowgraph (see Table 5.4). Tests 1-5 are the statement tests; tests 1-6 are the branch tests. However, from the list-patterns it can be concluded that the conditions for test 6 can never be met (a list-argument will always match one of the patterns [] or (xs:xss)). In general, infeasible paths can be introduced in the modelling phase as has been described in the previous section.

| | | | patterns and guards | | | | |
|---|---|---|---|---|---|---|---|
| test | expression | line | [ ] | [$x_1$,$x_2$] : xss | xs:xss | #xs=0 | #xs=1 |
| 1 | rect(0,0) | 41 | true | - | - | - | - |
| 2 | c $plus sumlist xss | 42 | false | true | - | - | - |
| 3 | sumlist xss | 44 | false | false | true | true | - |
| 4 | c $plus sumlist xss | 45 | false | false | true | false | true |
| 5 | sumlist ((take 2 xs):xss) | 46 | false | false | true | false | false |
| 6 | - | - | false | false | false | - | - |

*Table 5.4 Test cases for the function sumlist*

From the analysis of flowgraph and decomposition tree metrics, one may select functions which surpass certain pre-set threshold values, e.g. on testability or size. These functions can be inspected, and if necessary, they can be redesigned and implemented, resulting in more acceptable metric values. These threshold values may depend on the type of project in which the programs are going to be used. Functions which produce exotic primes in their flowgraphs (not D-structured) can be detected, and subsequent code inspection may reveal a bad programming style or error prone code.

In the previous section, a simple control flow model for Miranda function definitions has been described. Application of the model should reveal the need of further refinements of the model, such as expansion of multiple patterns, of composite guards and of the other expressions.

## 5.4 Dependency model

In this section, the callgraph model for Miranda programs is described. Four classes of functions will be distinguished:

- global functions: functions defined on the top level of the script
- local functions: functions defined within one of the top level functions, or defined within another local function
- primitive functions (or operators): these are in Miranda[14] the arithmetic operators (+, -, /, *, ^, div, mod), the boolean operators (&, \/, ~, =, >, <), the list operators (#, :, ++, --, !), and the function composition operator (.).
- library functions: functions defined in another script or in the standard library

[14] See the Miranda manual

A callgraph is a directed graph with nodes corresponding to the functions in a program, and edges corresponding to one function calling another. Multiple function calls are modelled with one edge in the callgraph. Primitive functions and calls to these functions are not included in the callgraphs.

In the callgraph, one may select any function as root node: a so called rooted callgraph is obtained, with all nodes of the callgraph (and corresponding edges) that are reachable from this root node. In the sequel, such a rooted callgraph with as root node the function f will be referred to as 'the callgraph from root f '.

The callgraph model has mainly been used for imperative languages, in tools such as Prometrix(1993). Contrary to for example programs in Pascal, in the usual Miranda programming practice, there is a heavy reliance on local functions. The number of local functions may easily surpass the number of top level functions with an order of magnitude. Even in a small example program as given in Table 5.1, there are local functions which may obscure the top level dependencies in the program. Therefore, two new classes of callgraphs will be introduced: the local callgraph and the global callgraph. The customary callgraph is partitioned in on one hand the global callgraph, with dependencies between the top level functions, and on the other hand local callgraphs for each top level function. Furthermore, larger programs are usually split up into several scripts. The dependencies between these scripts are modelled in the last class: the include callgraph.

Hence, the following four classes of callgraphs are distinguished:

- general callgraph: the customary graph with calls between the three type of functions (locals, globals and library functions)
- global callgraph: calls between top level functions and library functions (directly or indirectly via local functions)
- local callgraph: for each top level function, the calls between this function and other top level functions, library functions, and local functions which are in scope of the top level function in the root
- include callgraph: in this callgraph there are no function dependencies, but calls between scripts (via the include construct, see section 5.4.4)

Each of these classes of callgraphs will be discussed in turn in the following sections.

### 5.4.1 General callgraph

In the general callgraph the dependencies between the three classes of functions (local, global and library) are modelled. For example, in the general callgraph with as root the function main (see Figure 5.5), the global, local and library functions are:

- top level functions: main, sumlist, showct, plus, rect, im, re
- local functions: sublist defined in main (line 7), x in showct (line 32), y in showct (line 32), c in the second clause of sumlist (line 43), c in the third clause of sumlist (line 47), x in the previous mentioned function c (line 48)
- library functions: hd, shownum, take, drop

A function in a general callgraph will be denoted by the plain name of the function as it appears in the program. It is optional to include or to exclude the library functions. In Figure 5.5, the library functions are shown.

### 5.4.2 Global callgraph

In the global callgraph only top level functions are modelled, and dependencies with other top level functions possibly indirectly via local functions. In the global callgraph, a top level function will be denoted by the name of the function and a star-prefix, such as *main; library functions are denoted without star. In the global callgraph with as root the function main, the functions are:

- top level functions: main, sumlist, showct, plus, rect, im, re
- library functions: hd, shownum, take, drop

As with the general callgraph, it is optional to include or to exclude the library functions. In Figure 5.4, the library functions are not shown.

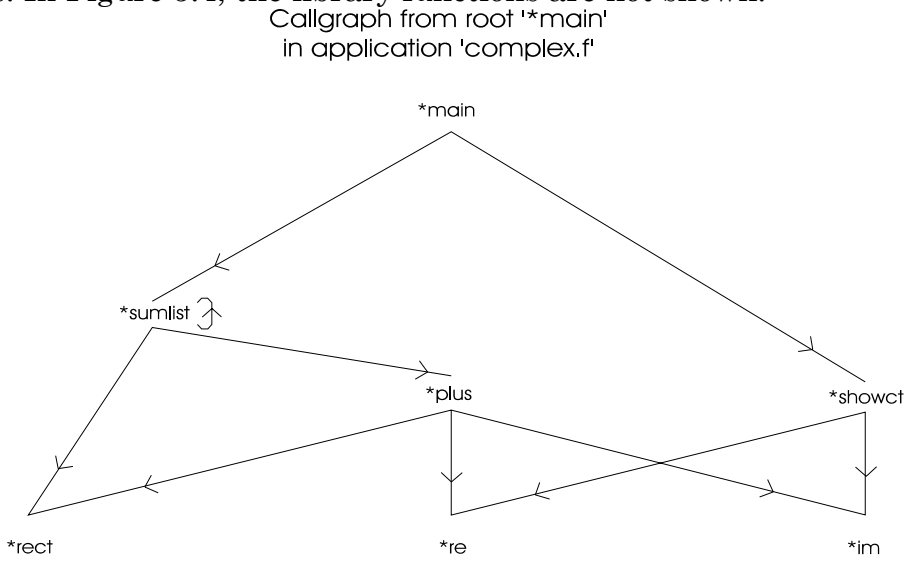



*Figure 5.4 The global callgraph from root main*

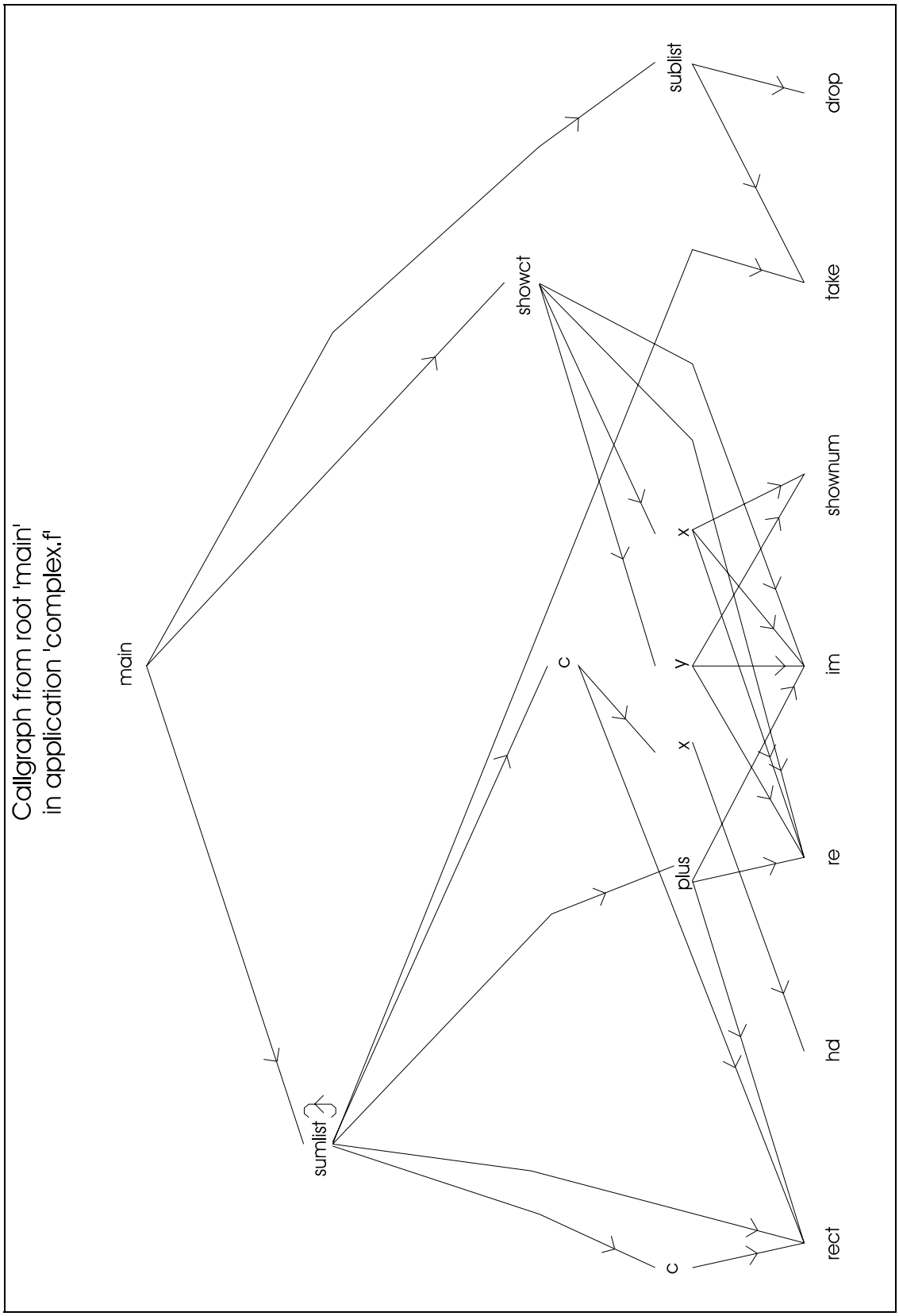



*Figure 5.5 The general callgraph from root main*

### 5.4.3 Local callgraph

In the local callgraph of a top level function, the dependencies of this top level function and the functions within the function definition are modelled. If another top level function is called in the function, the dependencies of that top level function are not part of the local callgraph. On this local level, these other top level functions are considered as 'library' functions.

In the local callgraph, the full name of the function will be used for the local function, i.e. the path will be the prefix of the name of the function as used in the script. The path consists of the global name of the function, the clause number in which the local function is defined, and so on, separated by a backslash. This full name allows the localisation of the clause in which the local function has been defined. In the local callgraph with as root the function sumlist (see Figure 5.6), the global, local and library functions are:

- top level functions: sumlist, showct, plus, rect
- local functions: c in the second clause of sumlist (\sumlist@2\c); c in the third clause of sumlist (\sumlist@3\c); x in the first clause of the function c in the third clause of sumlist (\sumlist@3\c@1\x)
- library functions: hd, take

Again, it is optional to include or to exclude the library functions. In Figure 5.6, the library functions are shown. The edge from \sumlist to sumlist implies a recursive call of the top level function in the root.

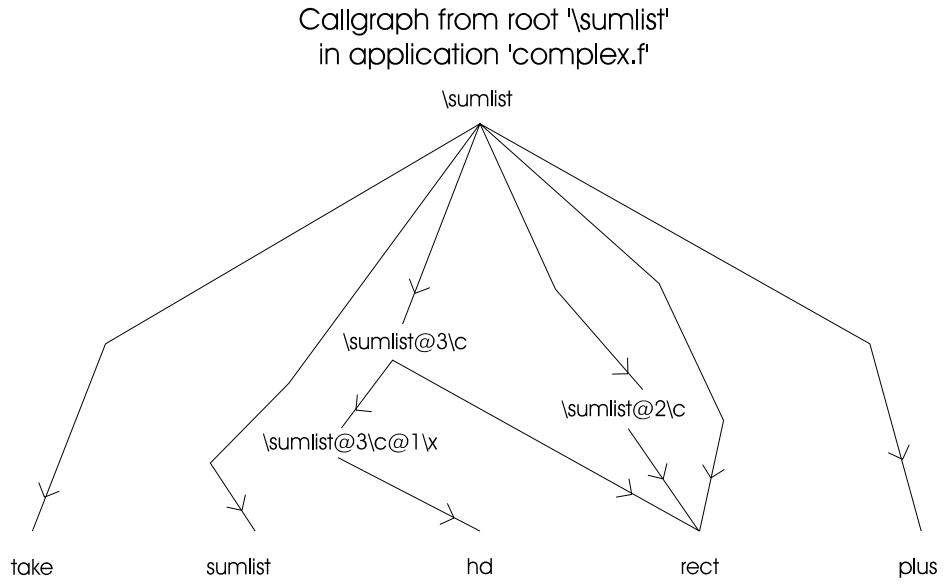



*Figure 5.6 The local callgraph from root sumlist*

### 5.4.4 Include callgraph

For large scale applications, a program is usually divided into several scripts. Functions defined in one script may be used in another script if the first script is included in the latter one. In imperative languages, e.g. Modula-2, this can be achieved by the IMPORT-declaration. In Miranda this is denoted by the construct *%include,* followed by the name of the file which contains the script. In the previous modelling, a function called from another script is considered as a library function.

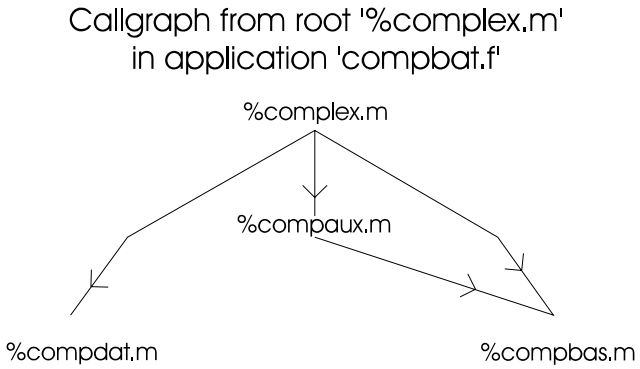



*Figure 5.7 The include callgraph from root complex.m*

From the include-constructs arises a hierarchy of scripts, which is modelled in the include callgraph (in the imperative domain called the module import graph; Pomberger, 1984). One abstracts from the actual calls to functions in the included script. The example program (given in Table 5.1) could be divided into four scripts (see Table 5.8): the file compbas.m with the base operations on complex numbers (line 11-32); the file compaux.m with the derived operations on complex numbers (line 33-48); the file compdat.m with the test data (line 8-10). The file complex.m only contains the main application (line 1-7). The include-constructs are added on lines 1a, 1b, 1c and 33a. The include callgraph with as root the script complex.m is given in Figure 5.7. The four edges correspond to the four include-constructs in the scripts.

### 5.4.5 Callgraph metrics

In this section, first some simple size metrics on callgraphs will be considered. The number of nodes and the number of edges will be used in the comparison of the general, the global and the local callgraphs. Then, other metrics on callgraphs will be given.

In Table 5.5 the number of the functions, i.e. the number of nodes in the graphs, are listed for the callgraphs with as root the function main, and the callgraphs with as root the function sumlist.

| rooted call-graph | # global functions | # local functions | # library functions | # functions total |
|---|---|---|---|---|
| main | 7 | 6 | 4 | 17 |
| *main | 7 | 0 | 4 | 11 |
| \main | 3 | 1 | 2 | 6 |
| sumlist | 5 | 3 | 2 | 10 |
| *sumlist | 5 | 0 | 2 | 7 |
| \sumlist | 3 | 3 | 2 | 8 |

*Table 5.5 Number of functions in callgraphs from root main and root sumlist*

For the four classes of functions described in section 5.4, there are six types of functions calls in callgraphs:

a. a global function calls another global function
b. a global function calls a local function
c. a global function calls a library function
d. a local function calls a global function
e. a local function calls another local function
f. a local function calls a library function

In Table 5.6 the number of the function calls, including recursive calls[15], are listed for the general callgraph with as root the function main, the global callgraph from root *main, and the local callgraph from root \main. The same properties are given for the function sumlist.

| # calls root | a. global-global | b. global-local | c. global-library | d. local-global | e. local-local | f. local-library | total |
|---|---|---|---|---|---|---|---|
| main | 10 | 5 | 1 | 6 | 1 | 5 | 28 |
| *main | 10 | 0 | 5 | 0 | 0 | 0 | 15 |
| \main | 3 | 1 | 0 | 0 | 0 | 2 | 6 |
| sumlist | 6 | 2 | 1 | 2 | 1 | 1 | 13 |
| *sumlist | 6 | 0 | 2 | 0 | 0 | 0 | 8 |
| \sumlist | 3 | 2 | 1 | 2 | 1 | 1 | 10 |

*Table 5.6 Number of function calls for callgraphs from root main and sumlist*

[15] In Prometrix (see section 5.5.1) recursive calls are not counted

As can be seen from the number of nodes and edges in these callgraphs, it is useful, for functional programs with many local functions, to obtain both the global callgraph and the local callgraphs, besides the customary callgraph.

For callgraphs, some standard metrics have been defined (Fenton, 1991; Prometrix, 1993). For each class of callgraphs introduced in the previous section, these metrics are applicable. For the general callgraph in Figure 5.5 with as root the function main, some of these metrics are given in Table 5.7. The definitions of the metrics are given in Fenton (1991). A short description will be given below.

| Metric | | Value |
|---|---|---|
| Size Metrics | | |
| – | Number of functions [16] | 17 |
| – | Number of function-function [16] paths | 27 |
| – | Volume | 44 |
| – | Average size | 2.59 |
| Dimensions | | |
| – | Maximum depth of calling | 4 |
| – | Minimum depth of calling | 4 |
| – | Fenton's width metric | 10 |
| Re-use Metrics | | |
| – | Reuse 1 metric | 0.65 |
| – | Reuse 2 metric | 0.61 |
| Impurity Metrics | | |
| – | Yin and Winchester C metric | 11 |
| – | Fenton's impurity metric (%) | 9.17 |

*Table 5.7 Metrics of the general callgraph from root main*

The volume is the sum of all sizes of functions, where each function's size is the number of nodes in its flowgraph. The minimum depth is the length of the shortest path from the root node to the farthest node in the graph. The maximum depth is the longest loop-free path between the root node and any other node. Fenton's width gives the maximum number of functions on any level. If no function is called more than once by one other function then there is no reuse. The callgraph is then a pure tree. The reuse metrics give the proportion by which the size of the program, in which functions are duplicated that are called from different places, exceeds the actual program.

[16] In the Miranda analyser (see section 5.5.1) the functions are referred to as modules

```
|| file complex.m                                                    1
%include "compbas"                                                  1a
%include "compaux"                                                  1b
%include "compdat"                                                  1c
|| main j k list  is the sum of the j-th through k-th              2
|| complex number in list                                          3
main ::  num -> num -> [[num]] -> [char]                            4
main j k list                                                       5
  = showct (sumlist sublist)                                        6
    where sublist = take (k-j+1) (drop (j-1) list)                  7
|| file compdat.m                                                   8
|| test data                                                        9
xs =  [[4,5],[1,0],[8],[],[2,3,4],[7,8]]                           10
|| file compbas.m                                                  11
|| specification complex numbers                                   12
|| re(rect(a,b)) = a                                               13
|| im(rect(a,b)) = b                                               14
                                                                    15
|| type definition complex numbers                                 16
abstype ct                                                          17
with                                                                18
   rect   :: (num,num) -> ct                                        19
   re     :: ct -> num                                              20
   im     :: ct -> num                                              21
   showct :: ct -> [char]                                           22
                                                                    23
|| implementation complex numbers                                 24
ct          == [num]                                                25
rect (a,b)  = [a,b]                                                 26
re [a,b]    = a                                                     27
im [a,b]    = b                                                     28
showct z    = x, if im z = 0                                        29
            = y ++ " i", if re z = 0                                30
            = x  ++ " + " ++  y  ++ " i", otherwise                 31
              where (x,y) = (shownum(re z), shownum(im z))          32
|| file compaux.m                                                  33
%include "compbas"                                                 33a
|| derived operations complex numbers                             34
plus        :: ct -> ct -> ct                                       35
c1 $plus c2 = rect (re c1 + re c2, im c1 + im c2)                   36
                                                                    37
|| sum of complex numbers in list                                 38
|| each complex number is derived from a list of numbers          39
sumlist               :: [[num]] -> ct                              40
sumlist []           = rect(0,0)                                    41
sumlist ([x1,x2]:xss)= c $plus sumlist xss                          42
                       where c = rect(x1,x2)                        43
sumlist (xs:xss)     = sumlist xss, if #xs = 0                      44
                     = c $plus sumlist xss, if #xs = 1              45
                     = sumlist ((take 2 xs):xss),otherwise          46
                        where c = rect(x,0)                         47
                                  where x = hd xs                   48
```

*Table 5.8 Example Miranda program with include files*

The Reuse 1 metric is based on the functions having equal weight; the Reuse 2 metric sizes each function according to the number of nodes in the flowgraph[17]. The impurity is the amount by which the graph deviates from a pure tree structure. The Yin and Winchester C metric is the callgraph equivalent of McCabe's metric and measures the number of calls 'branching in'. Fenton's impurity metric is a normalised measure, ranging from 0 (when the graph is a tree) to 100.

As with the control-flow metrics, one can detect functions that exhibit an extreme value on some metric. E.g., a high value of impurity metrics may point to a bad design, or if the design is good, to program code that strongly deviates from the design.

## 5.5 Miranda analyser

In previous research, an analyser has been constructed to obtain the Halstead and McCabe-metrics of Pascal programs and Miranda scripts (van den Berg, 1992). The implementation of this analyser was based on an attributed grammar in the Synthesizer Generator (1989). The present Miranda analyser for the flowgraphs and callgraphs is also devised on an attributed grammar. As back end of this analyser, the tool Prometrix (1993) is used. This system provides among others the graphical display of the graphs, the calculation of standard metrics on these graphs, and statistical analysis of the metrics. There are front ends available to this tool for several, mainly imperative, programming languages. The current Miranda front end, accomplishing the modelling described in the previous sections, is the first one for a functional programming language.

### 5.5.1 Prometrix

Prometrix (1993) is a tool for the analysis of callgraphs and flowgraphs. The three modes of operation are the following:

- The prepare mode: the front end for the respective programming language is invoked to produce the representation of the flowgraphs and the callgraph in an intermediate file (the .f file) for the given source code. From this file another file is produced with metric values (the .dat file). It is possible to process a number of scripts jointly (with the names of their files in a batch file), producing a single intermediate file with the representations of flowgraphs and callgraphs in all these scripts together.

[17] In Prometrix (see section 5.5.1) the number of nodes in library functions is taken to be 0. More appropriate would be the value 2, the size of a P1 flowgraph, considering a library function as an elementary action

- The inspect mode: both flowgraphs and callgraphs (from a .f file) can be displayed graphically, with their respective metric values. It is optional to display the library functions. One may select a subgraph, and it is possible to prune the graph, i.e. to contract dependencies in one node. One may alternate between nodes in the graphs and the related source code (code viewing). An example of a flowgraph is given in Figure 5.8, the flowgraph of the function sumlist (cf. the annotated flowgraph in Figure 5.2).
- The global mode: the metric values can be displayed in different formats (i.e. histograms, box plots).

For further details on the operation of Prometrix, the reader is referred to the manual.

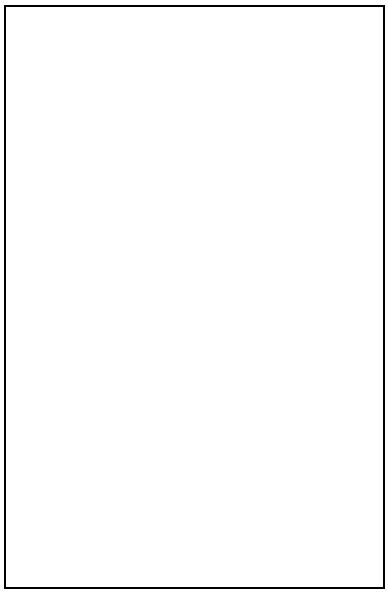


*Figure 5.8 Flowgraph of the function sumlist from Miranda analyser*

### 5.5.2 Miranda front end

The two main components of the front end are a pre-processor, and an 'editor' generated with the Synthesizer Generator (GrammaTech, 1993). The pre-processor has mainly the following functionality:

- to add semicolons to account for the offside layout rule in Miranda (Turner, 1986)
- to convert 'literate' Miranda scripts to 'normal' scripts (Turner, 1986)
- to calculate size metrics: the lines of code, with and without comments/white lines.

The editor derives the flowgraph and callgraph representations as attributes of the scripts. For each production in the abstract syntax, the attribute rules pro-

vide the contribution to the representations of the flowgraphs and the callgraphs. Two files are generated by the editor:

- a file (the .f file) with the standard flowgraph and callgraph representation. The metrics statistics (in the .dat file) are based on this file
- a file (the .f.f file) with the standard flowgraph representation and the general, global, local and include-callgraph representation[18]. Figure 5.4 toFigure 5.8 are examples of the output from the Miranda analyser.

For the flowgraphs and the general callgraphs, the names of the functions are encoded with their path: e.g. /sumlist@2/c@1/x. Prometrix will only show the part after the last slash. For the global callgraphs, the names of the functions are encoded with a star-prefix, e.g. *sumlist. For the local callgraph, the names of the functions are encoded with path and inverted slashes, e.g. \sumlist@2\c@1\x. Library functions are encoded with just their names as they appear in the program text. The names of the files in the include graph representation are encoded with a %-prefix. The files with the scripts are processed in a batch file (see the prepare mode of Prometrix in section 5.5.1).

### 5.5.3 Metric statistics

In the global mode of Prometrix, one can obtain the metric values in different formats. A part of a summary statistics table for the script complex.m (Table 5.1, without include files) is given in Table 5.9. The values of the maximum, minimum, mean, standard deviation and median for various metrics are given.

| Metric | Max | Min | Mean | Std.Dev | Median |
|---|---|---|---|---|---|
| Number of nodes | 11 | 2 | 3.07 | 2.43 | 2.00 |
| Number of edges | 15 | 1 | 2.71 | 3.77 | 1.00 |
| Biggest prime | 4 | 2 | -[19] | -[19] | 2.00 |
| Depth of nesting | 5 | 0 | 0.64 | 1.34 | 0.00 |
| McCabe's metric | 6 | 1 | 1.64 | 1.34 | 1.00 |
| State. testability | 5 | 1 | 1.43 | 1.12 | 1.00 |
| Branch testability | 6 | 1 | 1.64 | 1.34 | 1.00 |
| Fan-in | 4 | 0 | 1.57 | 1.35 | 1.00 |
| Fan-out | 6 | 0 | 2.00 | 1.73 | 2.00 |
| Fan-out ex. libraries | 5 | 0 | 1.57 | 1.64 | 1.50 |

*Table 5.9 Summary statistics of complex.m (without include files)*

[18] Aliases (see Miranda manual) are not taken into account

[19] This metric is on an ordinal scale, so this statistical quantity is not appropriate

The fan-in gives the number of functions that call a particular function; the fan-out is the number of functions that is called by a function. It is optional to include or exclude the calls to library functions. The number of functions defined in the example script (Table 5.1) is 14 (thus excluding the library functions); the total number of the nodes in the flowgraphs of these functions is 43. Figure 5.8 and Table 5.9 are examples of the output from the Miranda analyser.

The summary statistics provide a good objective basis for the comparison of different programs, for example in order to make a choice between competitive implementations with respect to the testability of the programs.

## 5.6 Design of functional programs

In the previous sections, the modelling and static analysis of programs in Miranda have been described. In this section, the analysis will be extended to designs of functional programs. Functional languages have been used in software development (Joosten, 1989) as executable specifications (Turner, 1985) and for prototyping (Henderson, 1986). Miranda programs can be developed in a top down manner by the use of stubs. The code can be analysed in the subsequent stages of the software development. Structured design in combination with prototyping in a functional language has been described by Harrison (Harrison, 1993a).

### 5.6.1 Pseudocode

The Miranda metric analyser described in section 5.5 can be used for designs with stepwise refinement in pseudocode as described below. The design callgraphs obtained in this way give the 'uses'-hierarchy as in structure charts (Yourdon & Constantine, 1979).

In the pseudocode, any Miranda language construct can be used, including the use of local definitions. However, the code need to be neither executable nor type-correct to the Miranda-system. The '=' symbol in the function definition denotes a 'uses' - relation.

The example problem of complex numbers from section 5.2 will be used again to illustrate the design of a Miranda program with this pseudocode. Two steps of refinement are given in Table 5.10.

### 5.6.2 Design callgraph

This design given above can be depicted in a structure chart (see Figure 5.9) without interface and procedural annotations.

The pseudocode can be offered to the Miranda metric analyser. The analyser will produce a design callgraph as in Figure 5.4, but now with the dependencies given in the structure chart of the design. The metrics defined on callgraphs in section 5.4.5 can be obtained for these design callgraphs as well.

```
|| A first design in pseudocode:

    main
    = getSublist
      convertAllToComplexList
      sumComplexList
      showComplex

    convertAllToComplexList
    = convertOneToComplex

|| A refinement of the first design:

    main
    = getSublist
      convertAllToComplexList
      sumComplexList
      showComplex

    getSublist list
    = drop firstpart list
      take secondpart list

    convertAllToComplexList list
    = [convertOneToComplex element | element <- list]

    convertOneToComplex list
    = complex(0,0), if length list = 0
    = complex(first list, 0), if length list = 1
    = complex(first list, second list), otherwise

    sumComplexList list
    = (head list) plus (sumComplexList (tail list))

    showComplex
    = showRePart
      showImPart
```

*Table 5.10 Design of example program in pseudocode*

From this second design, it is rather trivial to obtain a Miranda program. The data structures have to be chosen; the arguments of the functions have to be established; subsequently, the type declarations of the functions can be

given; and finally the remaining Miranda code of the functions. (Notice that this program will differ from the example program in Table 5.1.) Similar operations in the graph can be grouped together in one file, e.g. the operations on complex numbers. A modular structure will be obtained such as given in Table 5.8 (cf. Parnas, 1972).

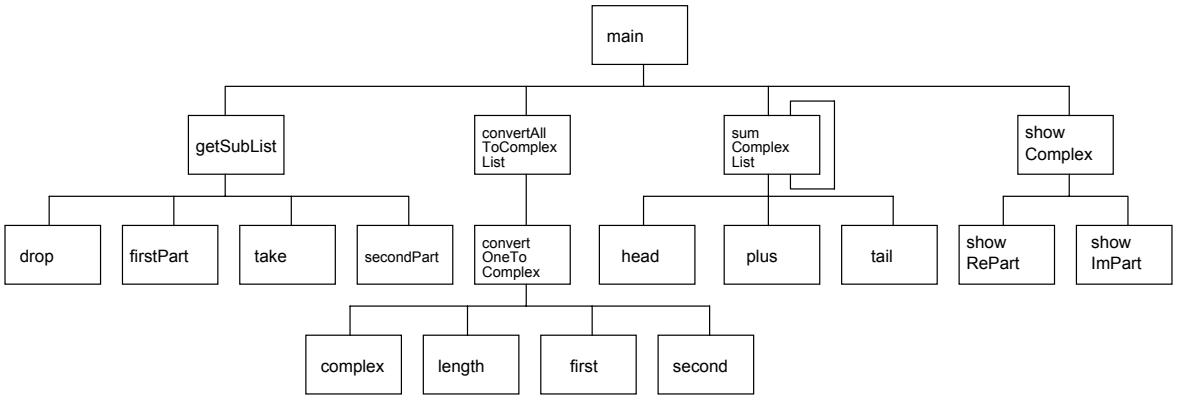



*Figure 5.9 Structure chart of design of example program*

The design callgraph metrics can be compared with the metrics of the callgraphs of the final program. Differences can be explained by details in the final coding, such as the use of auxiliary functions or local functions. However, there might have been other reasons to deviate from the design. In the example program in Table 5.1, the conversion of a list of numbers to a complex number is combined with the calculation of the sum of the list with complex numbers, resulting in a slightly more efficient program than the one obtained from the design above.

## 5.7 Conclusion

The metric analyser for Miranda programs is based on a flowgraph model and a callgraph model. The flowgraph model uses the top level control structure in the function definitions: the patterns, the guards and the expressions are not expanded. It is questionable whether a further expansion would be useful for the modelling aiming at the attribute of comprehensibility of programs. The present model allows the analysis of test cases, and the detection of error prone definitions written in a bad programming style. This hypothesis has to be tested in further experiments (van den Berg & van den Broek, 1995b).

The callgraph model in the analyser results in four classes of callgraphs. The include callgraph provides an insight in dependencies of the files used in the program. The global callgraph gives an abstraction of the dependencies of the top level functions in a script without being obscured by the local func-

tions. The local callgraphs are useful for a more detailed analysis of the individual top level functions. The general callgraph is used for the standard statistical analysis of the program.

Furthermore, the Miranda metric analyser allows the construction of structure charts on the base of a design in pseudocode. The metric values of these design callgraphs can be compared with the values obtained from the callgraphs of the final code. Differences may point to design decisions made in a later phase of the software development.

An important advantage of the modelling of functional programs presented here is the close similarity with the modelling used for imperative programs. The same standard metrics are applicable in both cases. In this respect, Harrison (1993b) showed a model that deviates from the modelling in section 5.4. For example, a function definition *f x = map h x*, is modelled by Harrison with two calls *(f, map)* and *(map, h)* instead of the calls *(f, map)* and *(f, h)*. However, the structure chart given in a previous article (Harrison, 1993a) is similar to the global callgraph introduced here.

Somewhat larger programs have been analysed with the Miranda analyser. Among others, a database system with about 800 lines of source code (not including comments), divided over 4 data files. (The program is roughly equivalent to about 8000 lines of imperative code (Turner, 1982)). There are about 450 functions defined in this system. Validation of the metrics, based on the flowgraph and callgraph model, has to be carried out for functional programs (cf. van den Berg *et al.,* 1993; van den Berg & van den Broek, 1994, 1995b). Furthermore, the analyser could be easily extended with a dependency graph of types that are defined in a script.

The metric analyser, with the Miranda front end to the Prometrix system, appears to be a very useful tool for the automated static analysis of also larger functional programs: by displaying the dependencies in callgraphs, for providing data on metric values of standard metrics on callgraphs and flowgraphs, and for detecting functions that are complex with respect to pre-set threshold values, e.g. size and testability.

## Acknowledgement

The authors would like to thank R. Bache for providing his notes on building front ends, A. Belinfante for his advise on the use of the Synthesizer Generator, B. Helthuis for his support in coupling the Miranda front end to Prometrix, M. Ramaer for the implementation of a part of the pre-processor, and D. van der Sar for his work on the dependency model.

# Part C : Validation

## Issues

Ince (1989) asserts that very little empirical validation of software metrics has occurred. What validation has been reported has been deficient in a number of respects. A major criticism is that the experimental design of metrics projects has been flawed. This is usually manifested in a sample size which is too small. Another criticism is that the sample of programmers or designers used has been artificial. Usually the subjects have been university students and not staff involved in serious software development. A further criticism is that much of the research carried out on metrics has ignored the large variation in ability that occurs in the subjects who have been studied. Finally, there is the criticism that reporting procedures can distort the validity of any experiment. He concludes as follows: A major activity over the next few years will be the empirical validation of metrics on *real* projects, with *real* staff, and in experiments which have been *properly* designed.

Schach (1990) distinguishes experimentation-in-the-small and experimentation-in-the-many, to denote experimentation in the areas of programming-in-the-small and programming-in-the-many, respectively. Experimentation-in-the-small is an acceptable scientific technique for determining the validity of a variety of software engineering techniques for programming-in-the-small. He states that: there is apparently no way of conducting acceptable [controlled] experimental trials to compare two [design] methods for programming-in-the-many.

In a discussion of toy versus real situations for experimental research, Fenton *et al.* (1994) conclude that evaluative research in the small is better than no evaluative research at all. A small project may be appropriate for an initial foray into testing an idea or even a research design.

Reviewing these issues, it has been stated by Shepperd and Ince (1994) that:

*The importance of validation cannot be overstressed: metrics based on flawed models are worse than valueless: they are potentially misleading. (Shepperd & Ince, 1994)*

## Experimental studies

Metrics have been used in the evaluation of the benefits of software engineering methods and tools. A critical review of experimental studies is given by Kitchenham *et al.* (1994). Three types of studies have been distinguished:

- formal experiments, i.e. a means of testing, using the principles and procedures of experimental design, whether a hypothesis about the expected benefit of a tool/method can be confirmed;
- case study, i.e. a trial use of a method/tool on a full scale project;
- survey, i.e. the collection and analysis of data from a wide variety of projects.

Each method has its advantages and its limitations. Formal experiments give a high degree of precision, but may not scale-up to 'real life'; case studies are of a realistic scale but may not generalise to other projects or other staff or other organisations; surveys are realistic and can be generalised, but it may be difficult to collect sufficient comparable data and to perform valid analysis.

In this review, some criteria are put forward for assessing the quality of the studies. The criteria for formal experiments are the following:

1. a well defined experimental hypothesis
2. full definitions of all treatments
3. response variables directly derivable from the experimental hypothesis
4. an experimental design that identifies and controls confounding effects
5. a full description of the experimental conditions
6. use of a defined statistical design
7. use of valid techniques of statistical analysis.

In Chapter 6, a study is described to explore the validation of structure metrics. In terms defined above, this study could be characterised as an exploratory formal experiment-in-the-small: it considers Miranda-type expressions to track the modelling, the definition of structure metrics, and the formal and experimental validation with respect to the comprehensibility of these expressions.

In Chapter 7, this study is extended in order to explore the application of the representational measurement theory to the validation of metrics.

*There seems to be a minimum assumption that the empirical relation system for complexity of programs leads to (at least) an ordinal scale. (Fenton, 1994)*

The assumption of the ordinal scale has been investigated both in the formal relational system and in the empirical relational system for Miranda type expressions. One important issue in this study is the use of axioms from measurement theory.

Chapter 8 reflects on the validation of software metrics as presented in the studies of the two previous chapters. As such, it raises more questions than it provides answers. The different types of validities are placed in the development process of a software metric. Again, the dilemma is discussed of the limited external validity of controlled experiments combined with their high internal validity, versus the high external validity of field studies combined with their low internal validity.

In Chapter 9, the findings of a more extended formal experiment are reported. The control-flow model for Miranda function definitions, described in Chapter 5, is used in the set-up of a controlled experiment on the comprehensibility of structured and nonstructured definitions. As in the previous experiments, students have been used in this experiment-in-the-small. The characteristics of formal experiments given above can be traced quite easily in this study. The experiment has been designed following a much cited study of Scanlan (1989). The chapter contains some criticism of this study. Furthermore, some counter-intuitive results are reported.

# Chapter 6

## 6. Validation of Structure Metrics: A Case Study [20]

*A framework for the validation of axiomatic structure metrics is presented. In a case study, the comprehensibility of type expressions in the functional programming language Miranda has been investigated. A structure metric for the comprehensibility of type expressions has been developed together with internal and external axioms. This structure metric has been validated experimentally. The calibrated metric function results in a good prediction of the comprehensibility.*

### 6.1 Introduction

Software metrics are used to quantify objectively attributes of software entities (Fenton, 1991). Three types of entities can be distinguished: products, processes and resources. Furthermore, there are two types of attributes: internal attributes and external attributes. The latter not only depends on the software entity, but also on other entities. Examples of internal attributes of software products are size and structure; maintainability and reusability are examples of external attributes. Structure metrics aim to quantify the internal structure of the product. A general theory of structure metrics is provided by Fenton & Kaposi (1989). Structure metrics are based on the compositionality principle.

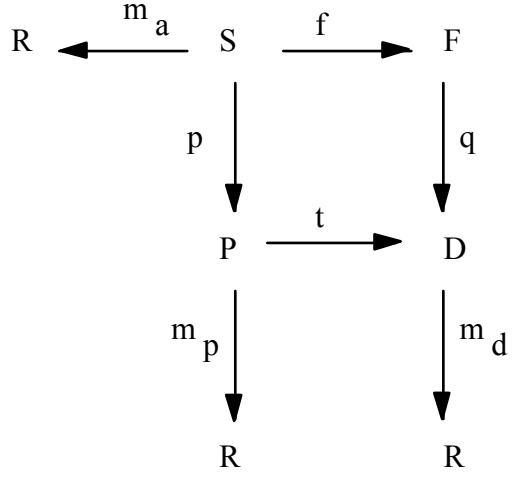
If
    $S, S_1, \ldots, S_n$: System
    $C$: System $\times \ldots \times$ System $\rightarrow$ System
    $m$: System $\rightarrow R$
then there exists a function $f_C$
    $f_C$: $R \times \ldots \times R \rightarrow$ System
such that
    $S = C(S_1, \ldots, S_n) \Rightarrow m(S) = f_C((m(S_1), \ldots, m(S_n))$

[20] K.G. van den Berg, P.M. van den Broek & G.M. van Petersen (1993). Validation of Structure Metrics: A Case Study. *Proceedings of International Software Metrics Symposium METRICS 93*, Washington: IEEE Computer Society Press, 92-99.

```
where
    R is a real number
    C is a system constructor
    m is a measure of attribute A
```

This principle asserts that the property of a system can be derived from the properties of its constituent components without knowledge of the interior structure of those components. Interaction between properties of components is excluded.

A scheme for the measurement of a software system is given in Figure 6.1 (cf. Melton, 1992). Structure metrics have been developed for computer programs in imperative programming languages.




| | | | |
|---|---|---|---|
| S | = system | R | = real number |
| f | = front-end | F | = flowgraph |
| q | = decomposition | D | = decomposition tree |
| p | = parser | P | = parse tree |
| t | = tree transform | $m_a$ | = metric on S |
| $m_p$ | = metric on P | $m_d$ | = metric on D |

*Figure 6.1 Scheme for measurement of software*

The control flow in a program S is modelled in a flowgraph F by a function f. By defining two constructors on flowgraphs, sequencing and nesting, a decomposition algorithm q yields a unique decomposition tree D. Consequently, a metric function $m_d$ can be defined on this tree structure resulting in a number R. Moreover, there is a non-structural measurement $m_a$ of the system with respect to attribute A. The order in the result of this $m_a$ on systems should correspond to the order on systems from the composition of functions $m_d \cdot q \cdot f$. A tool

supporting this analysis is Qualms (Bache & Leelasena, 1990). There are several front-ends for the modelling of programs in flowgraphs, i.e. the function f.

An alternative to this approach is using a grammar to define systems. A parse tree P is the result of the parser p of system S. Again, a metric function $m_p$ can be defined on this tree structure. The order in the result of this $m_a$ on systems should correspond to the order on systems from the composition of functions $m_p \cdot p$. The existence of a function t, which transforms a parse tree to a decomposition tree has to be investigated.

Grammars are used in complexity rankings of programs (Weyuker, 1988; Tian & Zelkowitz, 1992). The use of grammars is similar to the approach with algebraic structures as the base for compositional analysis (Zwiers, 1989). Algebraic specification has been used in the validation of software metrics (Shepperd & Ince, 1991). Attribute grammars have been used in software metrics (van den Berg, 1992)[21]. Structure metrics have been defined with attribute grammars (Whitty, 1992).

In the case study, the investigated software products are type expressions in the functional programming language Miranda. Type expressions and a structure metric are described in paragraph 6.3. The external attribute of these products is the comprehensibility to a human reader. The measurement of the comprehensibility will be described in paragraph 6.4. The general framework for the experimental validation is described in the following section.

## 6.2 A framework for validation

A scheme of the framework for the experimental validation of structure metrics is displayed in Figure 6.2. Some model, a flowgraph or a grammar, will be used to model the structure of the software product and results in a tree structure. The internal axioms provide the definition of a structure metric: this reflects the compositionality. The external axioms state the properties of the software entities and give the hypothetical order of these entities with respect to the external attribute.

The validation of the metric function consists of the following six steps:

a. The function satisfies the internal axioms: consequently, the function is a structure metric.

b. The function satisfies the external axioms: it provides a consistent measure with respect to the external attribute. This results in conditions on coefficients in the metric function.

[21] Chapter 3 of this thesis

c. The external axioms hold in practice: the actual order on software products, with respect to the external attribute, corresponds to the hypothetical order expressed in the external axioms.
d. The function is calibrated: the coefficients are given actual values, determined from a non-structural measurement of the software products with respect to the external attribute.

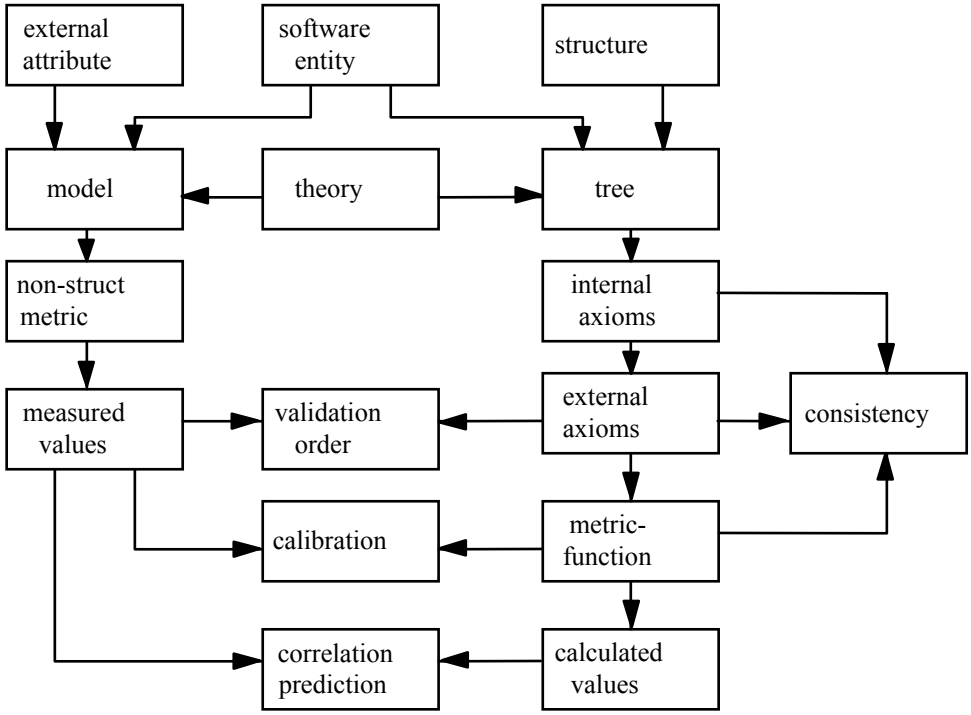



*Figure 6.2 Framework for the validation of structure metrics for an external attribute of a software entity*

e. The calibrated function is used for the rank order: the rank order correlation between measured and calculated values is determined.
f. The calibrated function is used for prediction (in stochastic sense): the efficiency of the prediction of actual values from calculated values is determined.

In the following section, the software entity in the case study - type expressions in the functional programming language Miranda - will be introduced. In addition, a structure metric for type expressions will be described.

## 6.3 Structure metrics of type expressions

In this paragraph, a subset of type expressions in Miranda, will be described. A grammar for this subset will be presented, followed by some alternative grammars. The internal axioms for a structure metric for type expressions will be given, and subsequently, the external axioms and the metric function itself.

### 6.3.1 Type expressions

Many programming languages provide some kind of typing system. In Modula-2, the type of variables has to be declared. The heading of a procedure declaration must contain the types of the parameters and the result. E.g., the function procedure

```
PROCEDURE Digit (K: CHAR): BOOLEAN
```

In the functional programming language Miranda it is optional to the programmer to provide the type of a function. The type-checker derives the type and compares this with the given type. The syntax of type expressions will be illustrated with some examples in Table 6.1.

```
digit :: char -> bool
The function digit returns True if the argument is
a digit and otherwise False

? digit '5'
True

head :: [*] -> *
The function head returns the first element of
a given list

? head [2,4,7,4]
2

first :: (*,**) -> *
The function first returns the first component of
a given 2-tuple

?first ('5',True)
'5'

split :: (*-> bool) -> [*] -> ([*],[*])
The function split returns, given a predicate (boolean
function) and a list, a tuple with the first component
the list with elements satisfying the predicate and
the second component the list with elements not
satisfying the predicate

? split even [2,4,7,4]
([2,4,4],[7])
```

*Table 6.1 Examples of type expressions with function applications in Miranda*

There are simple standard types, such as `char`, `bool` and `num`. The function constructor is denoted with an arrow →. The function `digit` has the type:

```
digit :: char → bool
```

Furthermore, there are type variables (Watt, 1990), in Miranda denoted with one or more stars. Structured standard types are lists, denoted with square brackets, and tuples, denoted with round brackets. In each example, the type of the function is given and an informal description. After the question mark prompt, a function application is given with its result on the next line.

### 6.3.2 A grammar for type expressions

Structure metrics for Miranda type expressions are derived from a grammar. The grammar for a subset of type expressions is given below: here, a Miranda data structure is used to model this grammar.

```
typeexp ::= Num | Bool | Char | Var num |
            L typeexp | T [typeexp] | F typeexp typeexp
```

The first line of the grammar gives the rules for the prime structures and the second line gives the rules for the three constructors in type expressions: the list constructor, the tuple constructor and the function constructor.

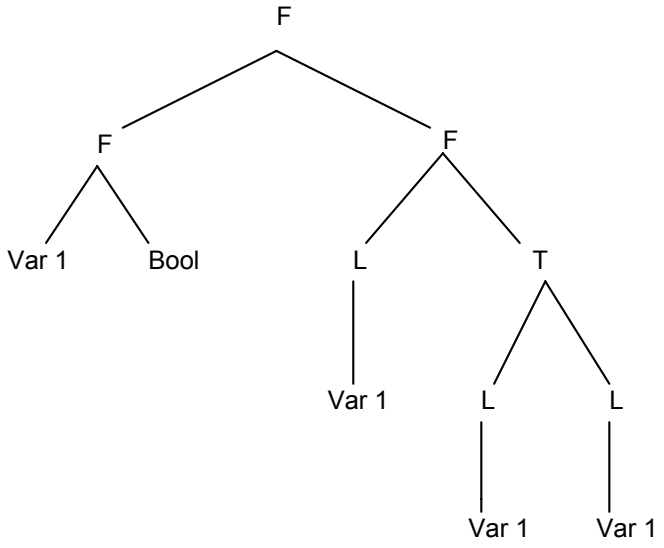



*Figure 6.3 Derivation tree of the type expression of the function* split

The type of the function `split` from Table 6.1

```
(* → bool) → [*] → ([*],[*])
```

can be parsed with this grammar, resulting in:

```
(F  (F (Var 1) Bool)
    (F (L (Var 1))
       T [L (Var 1)), (L (Var 1)]))
```

The parse tree or derivation tree of the function `split` is given in Figure 6.3.

### 6.3.3 Alternative grammars

The grammar described above is based on the right associativity of the function arrow. The type of the function `split` can be structured as a function with one argument (the predicate `(* → bool)`) and with as result a function with the type `([*] → ([*],[*]))`. This approach is named currying (cf. Watt, 1990). In other words, the type of `split` has been structured as follows:

```
(* → bool) → ([*] -> ([*],[*]))
```

There are two alternatives to this grammar. First, the type of the function can be structured as a function with one argument of the product type `((* → bool) × [*])` and with a result of type `([*],[*])`. The grammar for the function constructor in this case contains

```
F [typeexp] typeexp
```

The second alternative is obtained when the type of the function is structured in a similar way as the tuple: each function arrow separates types in the function constructor, in the same way as the comma separates the types of the components in a tuple.

The rule for the function constructor in this case is

```
F [typeexp]
```

Clearly, the derivation tree, and derived properties such as depth, depends on the chosen grammar. The ultimate choice of the grammar is determined by the psychological plausibility of the parsing model with respect to comprehension, and not by the actual parsing of the compiler. This approach has been used in the parsing of natural language sentences (Derivational Theory of Complexity (Fodor, Bever & Garret, 1974). New theories on the comprehension processes for natural languages point to shortcomings of this approach (McNamara, Miller & Bransford, 1991). One might expect interaction between properties of constituent components. However, this theory could be adequate for the hu-

man parsing of simple expressions in formal languages (cf. Green & Borning, 1990).

In the further validation study, the first grammar - based on the right associativity of the function arrow - has been used.

### 6.3.4 The internal axioms

A function m is a structure metric if it is defined according to the compositionality principle. For type expressions, a structure metric should satisfy the conditions listed in Table 6.2. These conditions are called the internal axioms. The first four axioms refer to the prime structures and the constants $c_i$ denote any number. The final three axioms refer to the constructors of type expressions.

| | |
|---|---|
| m(Num) | $= c_N$ |
| m(Char) | $= c_C$ |
| m(Bool) | $= c_B$ |
| m(Var n) | $= c_V(n)$ |
| m(L t) | $= f_L(m(t))$ |
| m(F $t_1$ $t_2$) | $= f_F(m(t_1),m(t_2))$ |
| m(T[$t_1$,...,$t_n$]) | $= f_T(m(t_1),...,m(t_n))$ |

*Table 6.2 Internal axioms for the structure metric of type expressions*

### 6.3.5 The external axioms

The order on software entities with respect to a certain attribute should be reflected in the values obtained by the metric functions. This order is described in an extension of the set of axioms, as has been done for flowgraphs (Fenton, 1991). These additional axioms are the hypotheses that will have to be tested. They will be referred to as the external axioms. The software entities in this case study are the Miranda type expressions, whereas the external attribute is the comprehensibility of these expressions.

Let t and $t_k$,... be type expressions. There are many possible hypotheses about the intuitive order, as will be seen below:

1. An order between the prime structures, e.g.:

   m(Var n) > m(Bool) > m(Char) > m(Num)

2. An order between type expression with a constructor and with its components, e.g.:
2.1. An order on (L t) and t

$$m(L\ t) > m(t)$$

2.2. An order on (F $t_1$ $t_2$) and $t_1$ and $t_2$

$$m(F\ t_1\ t_2) > m(t_1)$$
$$m(F\ t_1\ t_2) > m(t_2)$$
$$m(F\ t_1\ t_2) > max(m(t_1), m(t_2))$$
$$m(F\ t_1\ t_2) > m(t_1) + m(t_2)$$

2.3. An order on $T[t_1, \ldots, t_n]$ and $t_1 \ldots t_2$
There are similar hypothetical orders as on F, but generalised for the number of components. Some of these possibilities have been illustrated in Figure 6.4.

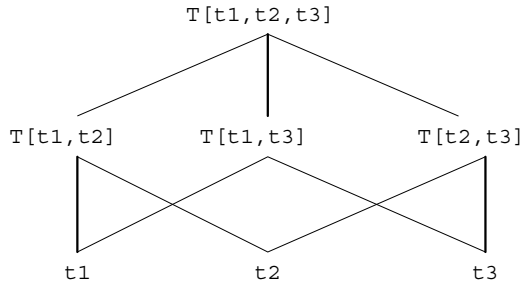



*Figure 6.4 Order of type expressions with the tuple constructor*

3. An order between type expressions with the same constructor, e.g.:

$$m(F\ t_1\ t_2) = m(F\ t_2\ t_1)$$
$$m(T[t_1, \ldots, t_{n+1}]) > m(T[t_1, \ldots, t_n])$$
$$m(T[t_1, \ldots, t_n]) = m(T[t_i, \ldots, t_j]),$$
$$\text{where } [t_i, \ldots, t_j] \in perms[t_1, \ldots, t_n]$$

4. An order between types with different constructors

$$m(F\ t_1\ t_2) > m(L\ t_i),\ i=1,2$$
$$m(T[t_1, t_2]) > m(L\ t_i),\ i=1,2$$
$$m(F\ t_1\ t_2) > m(T[t_1, t_2])$$

The external axioms on the comprehensibility as used in the case study are listed in Table 6.3. These hypotheses have been validated experimentally. This will be described in paragraph 6.4.

| | | |
|---|---|---|
| 1. | m(L t) | > m(t) |
| 2. | m(T[$t_1$,...,$t_n$]) | > max(m($t_1$),...,m ($t_n$)) |
| 3. | m(T[$t_1$,...,$t_n$]) | = m(T(perm[$t_1$,...,$t_n$])) |
| 4. | m(F $t_1$ $t_2$) | = m(F $t_2$ $t_1$) |
| 5. | m(T[$t_1$,...,$t_{n+1}$]) | > m(T[$t_1$,...,$t_n$] ) |
| 6. | m(T[$t_1$,...,$t_n$]) | > m(L $t_i$), i=1,...,n |
| 7. | m(F $t_1$ $t_2$) | > m(T[$t_2$,$t_1$]) |

*Table 6.3 External axioms for the structure metric with respect to the comprehensibility of type expressions*

### 6.3.6 The metric function

There are many candidates for the metric function on the tree structure, that has been obtained so far; e.g., there are the sum and product VINAP-measures (Fenton, 1991). In the Qualms system, many more metrics are available. From a compositional theory for the attribute, the actual choice can be made. However, the final choice will be determined by the performance of the metric function in a prediction system. For the type expressions in the case study, a simple sum metric has been chosen, as listed in Table 6.4.

| | |
|---|---|
| m(Num) | = $c_N$ |
| m(Char) | = $c_C$ |
| m(Bool) | = $c_B$ |
| m(Var n) | = $c_V$(n) |
| | |
| m(L t) | = $c_L$ + m(t) |
| m(F $t_1$ $t_2$) | = $c_F$ + m($t_1$) + m($t_2$) |
| m(T[$t_1$,...,$t_n$]) | = $c_T$ + m($t_1$) +...+ m($t_n$) |

*Table 6.4 Metric function m for the structure of type expressions*

This function m is consistent with the given internal axioms (Table 6.2). The constants $c_L$, $c_F$, $c_T$, $c_N$, $c_C$, $c_B$ and $c_V$(n) should fulfil certain conditions, which can be derived from the external axioms (Table 6.3) and the function definition (Table 6.4), in order that the function is in agreement with the external axioms.

In the following paragraph, the experimental validation of this structure metric for the comprehensibility of type expressions will be described. The actual values for the constants in the metric function m will be established.

## 6.4 Validation

The validation of structure metrics consists of six steps, as outlined in paragraph 6.2. The experimental validation of a structure metric for the comprehensibility of type expressions will be described now. The first two steps of the validation - the proof that the metric function satisfies the internal and external axioms - have been accomplished in the previous paragraph. The next four steps of the validation have to be carried out experimentally:
c. The external axioms hold in practice
d. The function is calibrated
e. The calibrated function is used for the rank order correlation
f. The calibrated function is used for prediction

Steps c and d are established in experiment 1 and steps e and f are verified in experiment 2. A detailed account of these experiments is given in van Petersen (1992).

For the (non-structural) measurement of comprehensibility of programs, there are several techniques known from literature (Robson, Bennett, Cornelius & Munro, 1991):
1. answering (multiple choice) questions
2. filling in blanks
3. writing a program for a given input and output
4. writing fitting input and output for a given program
5. modifying an existing program
6. localising errors in a program

In this case study, a variant of the third technique has been chosen. A type expression will be shown to the subject. He or she is requested to write a function, that will result in exactly this type when offered to the Miranda type checker. The time is measured between the moment that the type expression is shown to the subject until the answer is completed by the subject. For example, if the following type expression is shown

```
f  :: (num → char) → char → bool
```

then the following definition will be a correct answer:

```
f g 'a' = True, if g 5 = 'b'
```

It is not required that the function itself has any sensible meaning; just the given type must agree exactly with the type of the function obtained by the type checker.

### 6.4.1 Method

The subjects in the experiments are 16 first year students in Computer Science at the University of Twente. During one term, they followed a course in Functional Programming with Miranda (Joosten & van den Berg, 1990). They volunteered to the experiments and were randomly distributed over the two.

The material consisted of 40 questions with type expressions offered to the subjects. In each question the subject has to answer with a definition that matches with the given type expression.

### 6.4.2 Procedure

The questions are offered to the subjects on a system in the computer science laboratory (SUN workstations). The subjects know this system from their practical assignments in the programming course. First, there is a short introduction on how to answer the questions, and subsequently, two questions are presented for trial. With the standard editor (Vi) the subjects type their conceived answer. The time, elapsed between showing the question and leaving the editor, is measured automatically by the system. A countered balanced design is chosen in this experiment. All subjects get the same questions, but they are offered in a random order different to each subject.

### 6.4.3 Results

The hypothetical order on type expressions has been expressed in the external axioms. Each axiom has a left hand side (LHS) and a right hand side (RHS). The questions are assigned to the LHS and RHS of the axioms. In this way, questions can be used more than once. This approach is someway similar to the idea of atomic modifications (Zuse, 1991).

For example, axiom 1 states that `m(L t) > m(t)` . A question pair is for the LHS `[char → bool]` and for the RHS `char → bool`.

A within subject design is chosen. Per axiom and per student the average time is calculated for LHS-questions and RHS-questions. Only questions that belong to an axiom of which both sides are answered correctly are taken into account. Type writing errors in the answers have been corrected. The measured time is adjusted for an individual offset-time: the time for a subject to go with the cursor to the answer frame and leaving the editor, without giving an answer. Extreme values are discarded (in which the difference between the averaged LHS and RHS times for an axiom differs more than three times the standard deviation from the arithmetic mean). The differences between the LHS and RHS appear to have a normal distribution. The average time is calcu-

lated for each side and each axiom from the averages of all students. The results are given in Table 6.5.

The significance of the difference between the LHS and the RHS is calculated with the Fischer t test (which applies to differences between correlated pairs of means (Guilford & Fruchter, 1978). The degree of freedom is determined by the number of correct answer pairs (and not merely by the number of subjects). The results are shown in Table 6.5.

| | axiom | t_LHS (sec) | t_RHS (sec) | Fischer-t(n) |
|---|---|---|---|---|
| 1 | LHS > RHS | 19.0 | 08.0 | t(26) = +7.09* |
| 2 | LHS > RHS | 21.6 | 10.6 | t(27) = +5.08* |
| 3 | LHS = RHS | 33.8 | 29.7 | t(15) = +0.70 |
| 4 | LHS = RHS | 15.2 | 20.7 | t(22) = -3.12* |
| 5 | LHS > RHS | 25.6 | 20.5 | t(20) = +1.08 |
| 6 | LHS > RHS | 24.6 | 12.7 | t(25) = +4.08* |
| 7 | LHS > RHS | 19.7 | 12.7 | t(17) = +2.03 |

n = # degrees of freedom, * = p < .05

*Table 6.5 Results of the validation of the external axioms with respect to the comprehensibility of type expressions*

The measured values of the times for the good answers in the first experiment are also used for the calculation of the values of the coefficients in the metric function. The questions are grouped now per constructor. For example, the calculation of the coefficient $c_T$ from the equation:

```
m(T[Num, Bool, F Char Bool])
    = cT + m(Num) + m(Bool) + m(F Char Bool))
```

The time measured for the type expression left is 48 seconds and for the type expressions right is measured 5.0, 7.0 and 27.5 seconds respectively. From these values, $c_T$ has been calculated and averaged over the other values obtained for $c_T$. In Table 6.6 the average values for the coefficients are given.

| $c_L$ | $c_T$ | $c_F$ | $c_C$ | $c_N$ | $c_B$ | $c_V$ |
|---|---|---|---|---|---|---|
| 10 | 6 | 7 | 4 | 5 | 7 | 19 |

*Table 6.6 Values for the primes and the constants in the metric function (secs)*

In the second experiment, a new set of questions is offered to the second group of subjects. The average time is calculated from each good answer and, based on these values, the rank order of type expressions has been established.

Moreover, with the calibrated metric function from the first experiment, the rank order of the same type expressions has been calculated. The correlation between both orders has been determined according to Spearman (Guilford & Fruchter, 1978). The rank order correlation coefficient is 0.59. (Pearson's coefficient can not be used because the scores have been obtained in dependent pairs).

On the basis of the calculated value of the comprehensibility with the calibrated metric function, a prediction can be made of the actual comprehensibility (as would have been obtained by measurement). The forecasting efficiency (Guilford & Fruchter, 1978) is 19%; i.e. a reduction in variance of the predicted comprehensibility is achieved by using the calculated metric value.

## 6.5 Discussion

From the values in Table 6.5, it appears that for axioms 1, 2 and 6 there is a significant difference between the LHS and RHS, according the hypothesised order. For axiom 5 and 7, no significant difference has been found. In case of axiom 5, a possible cause of this fact could be that the expansion of a tuple with a component only gives a small, and therefore a difficult to measure, effect. For axiom 7, the reason of the small difference is not clear. Axioms 3 and 4 have to be treated separately. It seemed to be reasonable to include equalities in external axioms. However, equalities can not be validated experimentally in the way described before. Therefore, nothing can be concluded from the results for these two axioms.

It has been expected that the comprehensibility of a function is more difficult than of a tuple with the same components. Table 6.6 shows that the actual difference is small (but significant). The value for the constant $c_V$ for type variables is remarkably high.

The rank order correlation coefficient, for the calculated and measured values, is reasonably high. However, this coefficient is as high if the metric function returns the numbers of nodes and leaves in the decomposition tree. This case can be seen as the Halstead measure for the length of a 'program' (Halstead, 1977). The nodes are the total number of operators and the leaves the total number of operands. The so defined measure is a structure metric. It has not been checked whether this length metric satisfies all internal axioms. A high correlation coefficient is found as well if the calculated rank order is based simply on the size of the type expressions, i.e. the number of characters.

The forecasting efficiency is reasonably high. An even higher value (53%) is obtained when the values are grouped (16 groups). This leads to a considerable reduction in the variance of the prediction of the comprehensibility from the calculated metric value.

## 6.6 Conclusion

In this chapter a framework has been presented for the experimental validation of structure metrics. In a case study, a structure metric and its validation for the comprehensibility of type expressions in Miranda has been studied within this framework. No hard conclusions can be drawn about the absolute values obtained in the experiments. There is need for additional experiments. The validation for the alternative grammars of type expressions has to be carried out. Metric functions, which incorporate the depth of nesting, have to be investigated. The influence of type expressions for standard functions has to be included (e.g. the type expression `(* → *)` will be recognised as belonging to the standard identity function `id`). The set of primes has to be extended (e.g. a list of `char` will be comprehended as a string). The experiments have to be extended to include the whole Miranda type language (e.g. type synonyms, recursive types and abstract data types should be included). The effect of multiple occurrences of types in type expressions, which cannot be accounted for with compositionality, has to be investigated.

There are some general conclusions from this case study. The use of grammars, as an alternative to flowgraphs in the modelling of software in a tree structure, has been shown. The role of the external axioms, to express the hypothetical order on the software entities with respect to the external attribute, has been emphasised. In a prediction system based on structure metrics, there has to be a theory of compositionality for the external attribute. The experimental validation of the hypothetical order and the calibration of the metric function both require a large amount of experimental data on the software entities. Statistical analysis is needed to establish the actual order on these software entities and for the calculation of quantities, such as the rank order correlation coefficient and the forecasting efficiency.

## Acknowledgement

The authors would like to thank H. Muijs for the modelling of the Miranda type expressions as part of his M.Sc. thesis, and J. van Merriënboer for valuable comments on the experimental set up and the statistical analysis.

*Zwei geordnete Mengen* M *und* N *nennen wir* 'ähnlich'*, wenn sie sich gegenseitig eindeutig einander so zuordnen lassen, dass wenn* $m_1$ *und* $m_2$ *irgend zwei Elemente von* M*,* $n_1$ *und* $n_2$ *die entsprechenden Elemente von N sind, alsdann immer die Rangbeziehung von* $m_1$ *zu* $m_2$ *innerhalb* M *dieselbe ist, wie die von* $n_1$ *und* $n_2$ *innerhalb* N*. Eine solche Zuordnung ähnlicher Mengen nennen wir eine 'Abbildung' derselben auf einander.*

*Cantor (1895)*

# Chapter 7

## 7. Axiomatic Testing of Structure Metrics [22]

*In this chapter, axiomatic testing of software metrics will be described. The testing is based on representation axioms from the measurement theory. In a case study, the axioms are given for the formal relational structure and the empirical relational structure. Two approaches of axiomatic testing are elaborated: deterministic and probabilistic testing.*

### 7.1 Introduction

In this chapter, axioms from representational measurement theory will be utilised to establish the theoretical and empirical order of software entities with respect to some attribute. A simplified model for software measurement will be used (see Figure 7.1). Software entities will be considered: products, processes or resources (Bush & Fenton, 1990). Data on an external attribute (e.g. maintainability, reusability) of these entities are collected with some measure m'.

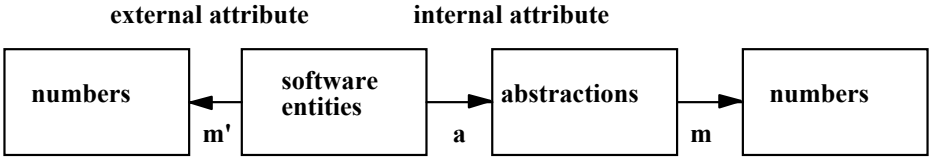



*Figure 7.1 Model for software measurement*

The external attribute will be related with some internal attributes, such as size or structure. The internal attribute is measured with a metric function m on abstractions of the software entities. The measure m is validated to the extent to which it preserves the order on the software entities obtained independently of m by the quantified criterion m' (Melton *et al.*, 1990).

---

[22] K.G. van den Berg & P.M. van den Broek (1994), Axiomatic Testing of Structure Metrics. *Proceedings of the Second International Software Metrics Symposium*, London: IEEE Computer Society Press, 45-53.

In a case study, axioms from the measurement theory will be tested, both formally and empirically. The case itself, comprehensibility and structure of type declarations, is of interest to researchers in the field of programming methodology. More general, the case is used to exemplify the application of representational measurement theory in software measurement and validation.

The representational approach has been used in software measurement (e.g., Baker *et al.*, 1990; Fenton, 1990; Melton, 1990; Bieman *et al.*, 1992; Melton *et al.*, 1992; Zuse, 1992). Some basic concepts in this measurement theory (Krantz *et al.*, 1971; Finkelstein & Leaning, 1984; Suppes *et al.*, 1989; Luce *et al.*, 1990) will be defined according to Roberts (1979).

A *relational structure* is a (n+1)-tuple $(A, R_1,...,R_n)$, where A is a set, and $R_1,...,R_n$ are relations on A. A function f: $A \rightarrow A'$ is called a *homomorphism* from relational structure $(A, R_1,...,R_n)$ into relational structure $(A', R_1',...,R_n')$ if, for each $i \in 1..n$,

$$R_i(a_1, a_2, \ldots, a_{ri}) = R_i'(f(a_1), f(a_2), \ldots, f(a_{ri}))$$

A homomorphism is an order preserving mapping. The triple $((A,R_1,...,R_n), (A', R_1',...,R_n'),f)$ is said to be a *scale*. If $(\mathcal{A},\mathcal{B},f)$ is a scale and $\varphi$ is a function such that $(\mathcal{A},\mathcal{B},\varphi.f)$ is a scale as well, then $\varphi$ is said to be an *admissible transformation of scale*. The *representation* $\mathcal{A} \rightarrow \mathcal{B}$ is *regular* if all scales $(\mathcal{A},\mathcal{B},f)$ are related via an admissible transformation of scale. The class of admissible transformations of scale of a regular representation defines the *scale type* of the representation. Some common scale types are: absolute, ratio, interval, ordinal and nominal scale (Roberts, 1979: 64). The focus in the case study is on *ordinal* scales, which are defined by monotone increasing transformation functions.

The aim of software measurement is to enable the comparison of software entities with respect to some attribute. As given in the model of Figure 7.1, there are four relational structures. The outmost structures are numerical relational structures. The software entities with their relations form the empirical relational structure. The fourth relational structure involves the abstractions. Axioms, that will be tested, state sufficient conditions for the existence of a regular scale. By investigating the axioms, the representation and measurement scale of these structures can be established.

In order to make the discussion of axiomatic testing concrete, a case study will be presented related to a specific kind of software documentation: type declarations. The programmer provides explicit information about the type of the objects in the program. This form of documentation not only may have an impact on the reliability of the software, but also on the comprehensibility to human readers of the programs (reviewers, maintenance programmers). The

software entities are type declarations in the functional programming language Miranda (Turner, 1986). Type declarations themselves have a certain degree of (cognitive) complexity: they are easy or difficult to comprehend. 'The comprehensibility' will be taken as the external attribute. The internal attribute is 'the structure' of type expressions. The relationship between the comprehensibility of type expressions and their structural properties will be investigated: first, by establishing the scale of measurement of the internal attribute and the external attribute, and then by investigating the correspondence between both measurements.

This chapter is organised as follows. In section 7.2, more details about the software entities in the case study, the type declarations, will be given. In the subsequent section, the modelling of the structure of type expressions is elaborated. The actual collection of data on the external attribute, the comprehensibility, will be described in section 7.4, with an exemplification of the deterministic and probabilistic testing of axioms. The final section discusses some results obtained with this approach.

## 7.2 The case study

The software entities considered in the case study, type expressions, will be introduced. In an imperative programming language like Modula-2 or Pascal, the heading of a procedure declares the type. For example, the heading of the function procedure *IsDigit* is:

```
PROCEDURE IsDigit (C: CHAR): BOOLEAN;
```

In the case study, type expressions in the functional programming language Miranda are considered. The type declaration of the function *isdigit* is (denoted with :: and on the right hand side a type expression):

```
isdigit :: char → bool
```

A function type is denoted with the symbol "→". The function *split* has a more complex type: *split* returns a tuple with two lists of numbers, one with elements of a given list that satisfy a predicate, and the second list with elements that do not.

```
split :: (num → bool) → [num] → ([num],[num])
```

The type of the predicate, the first argument of the function *split*, is a function type (num → bool). This argument is enclosed by round grouping brackets. The

type of the second argument [num] is a list type, a list of elements of type num. A list type is denoted with square brackets. The type of the result of the function is a tuple type with two components, each of type [num]. A tuple type is denoted with round brackets and component types separated by a comma. It is possible to define type *synonyms*, e.g. numlist == [num]. The type of function *split* with this synonym is:

```
split :: (num → bool) → numlist → (numlist,numlist)
```

The canonical form of the two given types of the function *split,* as used by the type checker, is the same (and equal to the first one).

Type declarations form an important clue to the understanding of functions in a program. They give a partial specification of the function: the type of its arguments and the type of the result. The complexity of the type declaration might give an indication of the complexity of the task to be performed by the function (e.g. Cardelli & Wegner, 1985).

In the 'real world model' (Maki & Thompson, 1973), restrictions will be imposed on the 'real world' entities and phenomena. In the case study the type expressions will be restricted to three standard types: *char*, *num* and *bool*, and three type constructors: the function type, the tuple type, and the list type (for homogeneous lists). Furthermore, neither type variables nor abstract data types are considered. Also, the influence of the naming of types and typographic issues will not be considered. Type synonyms are not considered. In other words, these aspects will be kept invariant in the case study. Type expressions with these restrictions will be called *simple* type expressions.

Type expressions are studied in the context of programs developed in an academic environment. It is evident that comprehensibility depends on the experience of the reader. The case study is carried out with novice Miranda programmers with corresponding proficiency. Only structural properties of simple type expressions in Miranda in relation with their comprehensibility to novice programmers are examined.

## 7.3 The theoretical order

In this section, the modelling of type expressions is described. The abstraction of type expressions is defined, a relation on abstract type expressions and a structure metric (cf. Melton *et al.,* 1990). Structure metrics are based on the compositionality of the structural properties (Fenton & Kaposi, 1989). On the basis of these definitions, the scale of measurement is established.

### 7.3.1 The abstraction

A relational structure $(A, R_1,...,R_n)$ is defined. Set A consists of abstract type expressions; $R_1,...,R_n$ are relations on abstract type expressions. In some cases, the corresponding operation of a relation will be used in the relational structure (cf. Roberts, 1979: 41). These operations are called *concatenation operators* or *constructors*.

First, the mapping of simple type expressions to abstract type expressions is defined. This *abstraction* implies the following rules:

1. the order between components in a tuple type expression is disregarded
2. grouping brackets around a tuple type expression with one component are disregarded
3. grouping brackets implied by the right associativity of the function type constructor are disregarded.

For abstract type expressions the following data structure will be used as model:

```
texp ::= L texp | F [texp] | T {texp} | C | N | B
```

with respectively: L the list type constructor; F the function type constructor; T the tuple type constructor; C the standard type *char*; N for *num* and B for *bool*. Next to the constructors, [texp] denotes an ordered list of abstract type expressions, and {texp} denotes a multiset. Some examples of the abstraction are given in Table 7.1.

| | type expression | abstraction |
|---|---|---|
| $t_a$ | (**(**num → [bool]**)**, bool) | T { F [N, L B], B} |
| $t_b$ | (bool, num → [bool]) | T { B, F [N, L B]} |
| $t_c$ | num → **(**[bool] → bool**)** | F [ N, L B, B] |
| $t_d$ | num → [bool] → bool | F [ N, L B, B] |
| $t_e$ | **(**num → [bool]**)** → bool | F [ F [N, L B], B] |
| $t_f$ | (num→bool)→[num]→([num],[num]) | F [ F [N,B], L N , T {L N, L N}] |

*Table 7.1 Example type expressions with abstractions*

The abstractions of $t_a$ and $t_b$ are the same: the round brackets in (num → [bool]) are disregarded (rule 2), as the order of components in the tuple (rule 1). The abstractions of $t_c$ and $t_d$ are the same. The abstraction of $t_c$ is not F [ N, F [ L B, B]], since the round brackets in ([bool] → bool) are disregarded (rule 3). In other words, the type of the result of a function is not allowed to be a function type. The abstractions of $t_d$ and $t_e$ are different, since the grouping

brackets in $t_e$ can not be disregarded ($\rightarrow$ is right associative). With $t_f$ the abstraction of the type of the example function *split* is given.

There are alternative abstractions; for example, to restrict the function type to F [$t_1$, $t_2$] (only two types in a function type); or to disregard the order of the arguments of functions. These alternatives are discussed in van den Berg *et al.* (1993). The choice between abstractions of entities is determined by the actual use of the abstractions: the establishment of a good correspondence between an internal attribute based on these abstractions, and an external attribute of the entities.

### 7.3.2 The containment relation

The containment relation on abstract type expressions, denoted by $\prec$, will be defined. Let a and b be abstract type expressions, with concatenation operators $\oplus$ and $\otimes$ respectively, and with maximal subexpressions $a_1,...,a_n$ and $b_1,...b_m$ respectively, as depicted in Figure 7.2.

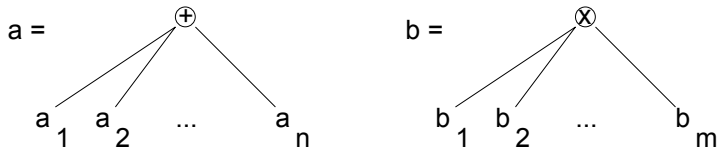



*Figure 7.2 Two abstract type expressions*

Then a $\prec$ b iff a is contained in b in the following sense: a is contained in b if a is contained in $b_i$ for some i. Moreover, if $\oplus = \otimes$ then a is contained in b if each $a_i$ is contained in some $b_{i'}$, subject to the conditions that 1',...,n' are pairwise different, and if $\oplus$ = F then [1',...,n'] is ordered. The containment relation is a partial order.

Consider, for example, the set of abstract type expressions in Figure 7.3:

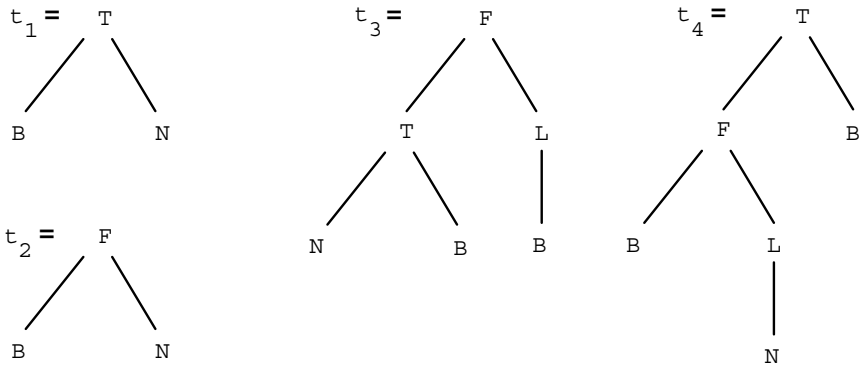



*Figure 7.3 Example abstract type expressions*

The containment relation of these type expressions, ($\{t_1,t_2,t_3,t_4\}$, $\{(t_1,t_3), (t_1,t_4), (t_2,t_4)\}$), is given in partial order diagram (Hasse diagram) in Figure 7.4.

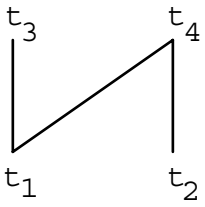



*Figure 7.4 Partial order with $\prec$ on example type expressions of Figure 7.3*

With the definition of this containment relation, the formal relational structure (texp, $\prec$, L, F, T, C, N, B) for abstract type expressions has been described.

### 7.3.3 Extension of the containment relation and ordinal scale

For the relational structure from the previous section, a measurement scale will be considered, based on theorems from measurement theory (Roberts, 1979). Only ordinal measurement will be discussed here. The following theorem will be used:

Suppose A is a countable set and R is a binary relation on A. If f is a real-valued function on A which satisfies

$$a\ R\ b \quad \Leftrightarrow \quad f(a) \le f(b) \qquad (.\ 1)$$

then ((A, R), (Re, ≤), f) is an ordinal scale (Roberts, 1979: 110).

For abstract type expressions, a linear *structure metric* function m is defined, with all constants $c_i \ge 0$ :

$$\begin{aligned}
m(C) &= c_C && (.\ 2)\\
m(N) &= c_N && (.\ 3)\\
m(B) &= c_B && (.\ 4)\\
m(T\{t_1,\ldots,t_n\}) &= c_T + m(t_1) + \ldots + m(t_n) && (.\ 5)\\
m(L\ t) &= c_L + m(t) && (.\ 6)\\
m(F[t_1,\ldots,t_n]) &= c_F + m(t_1) + \ldots + m(t_n) && (.\ 7)
\end{aligned}$$

The theorem above is not applicable for this function m and the containment relation $\prec$ on type expressions, because equation 1 is not satisfied ($\prec$ is a partial order). Therefore, with this function m a new relation $\prec_m$ on type expressions is defined as follows:

$$t_a \prec_m t_b \quad \Leftrightarrow \quad m(t_a) \le m(t_b) \qquad (.\ 8)$$

The relation $\prec_m$ is an *extension* of the containment relation $\prec$, i.e.

$$t_a \prec t_b \quad \Rightarrow \quad t_a \prec_m t_b \qquad (. 9)$$

From the theorem above it follows that ((texp ,$\prec_m$), (Re, ≤), m) is an ordinal scale.

In this section, an abstraction of type expressions and a containment relation on abstract type expressions have been defined. An extension of this relation derived from a structure metric function provides measurement of the internal attribute structure of type expressions on an ordinal scale. This allows the investigation of a correspondence of the extension with the empirical order as given by the quantified criterion, which also maps on (Re,≤) (see subsequent section). This approach differs from the one proposed by Fenton (1992). For a partial order on flowgraphs, Fenton defines a mapping of flowgraphs to (N,|), where N is the set of natural numbers and | is the relation 'divides without remainder', instead of a mapping to (Re,≤), in order to satisfy the representation condition (equation 1). In the following section, the empirical order of type expressions will be discussed.

## 7.4 The empirical order

In this section, the order of type expressions will be established with respect to the external attribute comprehensibility. The conditions for an ordinal scale are investigated.

There are several approaches to the measurement of comprehensibility of programs. In the case study, one measure has been chosen for the comprehensibility of type expressions (van den Berg *et al.*, 1993)[23]: the time in seconds needed for a subject to read a given type expression and to conceive and typewrite a (function) definition with exactly this type in the 'standard' programming environment. The time between showing the type expression on the screen and the completion of the answer is measured automatically. Afterwards, with the type checker of the programming system, the answer is marked as correct or incorrect. This time measurement will be used as criterion for the comprehensibility.

The data have been collected in controlled experiments. The subjects in the experiment are novice programmers, all first year students in computer science. Two data sets are used, each based on responses of 14 subjects to 42 type expressions (per data set 588 responses). Dataset1 consists of responses of 14

[23] Chapter 6 of this thesis

subjects to 16 type expressions, with a total of 241 correct responses; dataset2 is based on responses of 14 other subjects to another set of 16 type expressions, with a total of 347 correct responses. The type expressions are offered to the subjects in random order. Of the 42 questions in the original experiment, expressions with type variables have not been considered here, neither have questions with less than 6 correct answers. In Table 7.2 the type expressions of dataset1 are given. Further details of the experiments can be found in (van Petersen, 1992; van den Berg *et al.*, 1993).

| rank | nr | type expression | # correct (max 14) | average time (sec) | standard dev (sec) |
|---|---|---|---|---|---|
| 16 | 20 | bool → char → bool | 6 | 50.0 | 27.6 |
| 15 | 27 | (num,bool) → (num,bool) | 10 | 44.7 | 24.7 |
| 14 | 12 | num → char → char | 7 | 35.0 | 19.6 |
| 13 | 13 | [(num,bool)] | 12 | 30.0 | 9.8 |
| 12 | 18 | bool → num | 14 | 28.9 | 12.8 |
| 11 | 5 | [char] | 12 | 24.8 | 6.2 |
| 10 | 34 | (num,bool,char) | 13 | 24.7 | 7.2 |
| 9 | 15 | num → bool | 12 | 23.3 | 9.5 |
| 8 | 28 | char → char | 7 | 23.1 | 6.7 |
| 7 | 17 | char → bool | 8 | 21.8 | 8.3 |
| 6 | 26 | num → num | 10 | 19.7 | 10.6 |
| 5 | 14 | (num,bool) | 14 | 18.8 | 5.4 |
| 4 | 3 | bool | 14 | 17.7 | 9.6 |
| 3 | 2 | num | 14 | 14.6 | 5.5 |
| 2 | 41 | (num,num) | 13 | 13.9 | 3.1 |
| 1 | 1 | char | 13 | 12.8 | 3.2 |

*Table 7.2 Ranking of type expressions in dataset1 according to the average time*

The following approaches in the analysis of the data will be used. Firstly, a global analysis will be given based on the average time measured for each type expression. Secondly, an axiomatic analysis of the relative preference of each subject between pairs of type expressions will be described. Finally, an axiomatic analysis based on the relative frequencies of these preferences will be considered.

### 7.4.1 Global analysis of the empirical order

For each type expression, the average time for all correct responses has been calculated. The data for the first set are given in Table 7.2.

In Figure 7.5, for a subset of type expressions of dataset1, the empirical order based on the average time and the theoretical partial order are compared. The empirical order for this subset, except the value obtained on type expression 41, is an extension of the partial order of the 9 abstract type expressions. Taking into account a rather large standard deviation in the measured values, there is reasonable agreement between the theoretical order and the empirical order based on the average time. However, the scale type of the empirical order itself is not yet known from this analysis. For this purpose, the properties of this order have to be analysed by examining the axioms, as has been done for the theoretical order in the previous section.

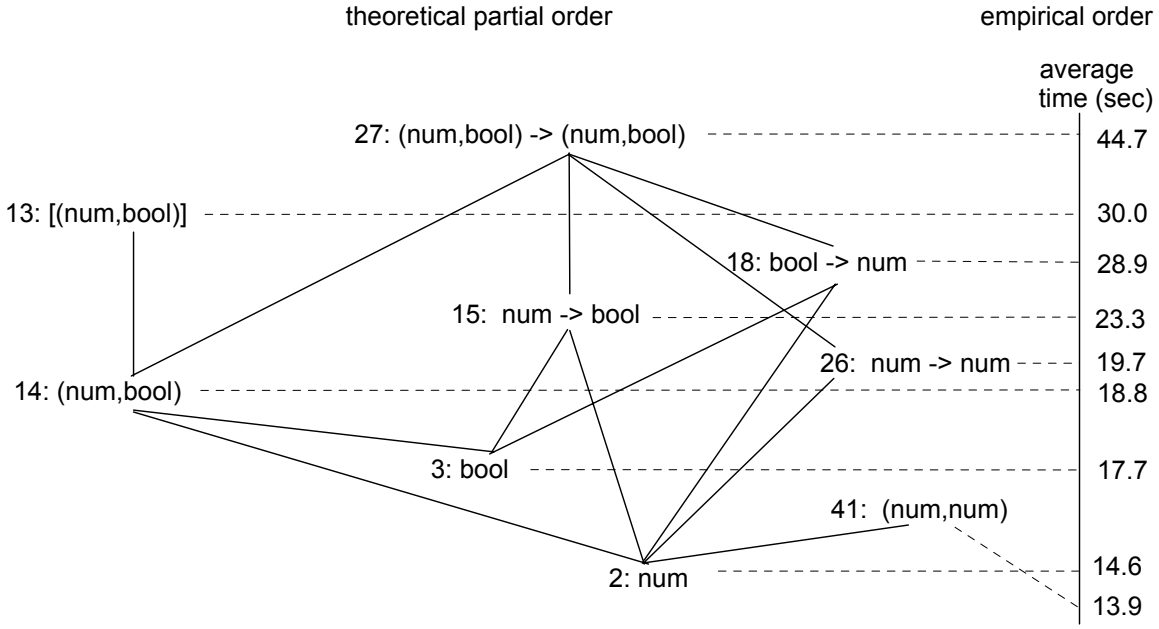



*Figure 7.5 Theoretical partial order in Hasse diagram and empirical order of subset of type expressions in dataset1*

### 7.4.2 Axiomatic analysis of the empirical order

Two types of axiomatic analysis will be ensued: a deterministic analysis and a probabilistic analysis. Each of them aims at establishing the representation of the empirical order by testing the axioms from the theorems. The theory of the deterministic analysis can be found in Krantz *et al.* (1971); of the probabilistic analysis in Suppes *et al.* (1989). In this section, Roberts (1979) will be used as the main reference. It should be expected that the comprehensibility measure in the experiment is on an ordinal scale. In that case the data should be conform a (strict) weak order.

### 7.4.2.1 Deterministic axiomatic analysis

On the basis of the time measurement (in seconds) for each type expression per subject, it is possible to define the relation R for all type expressions a, b in the data set A: aRb $\Leftrightarrow t_a > t_b$. This relational structure (A,R) represents the 'preference' of each subject in the indication of the most difficult type expression. The preference structure (A,R) can be represented in the *preference matrix* (A,p) defined as, $\forall$ a,b $\in$ A:

$$p_{ab} = 1 \Leftrightarrow aRb \text{ and } p_{ab} = 0 \Leftrightarrow \neg\, aRb \qquad (. 10)$$

The ranking of correctly answered questions per subject is determined. All these individual rankings form a profile, i.e. a list of k rankings (k is the number of subjects). In the experiment, not all individual rankings are complete, since not all questions have been answered correctly. There are only 2 subjects for each data set with a complete ranking. A reduction of dataset1 to a subset of 7 questions (subset1 = {12, 13, 15, 18, 20, 27, 34}) results in 5 complete rankings; also, a reduction of dataset2 to 7 questions (subset2 = {115, 118, 123, 124, 127, 129, 132}) results in 5 complete rankings.

A *group preference structure* (A,M) from a list of complete individual preference structures can be derived, for example according to the *simple majority rule,* defined as follows (Roberts, 1979: 118):

$$aMb \Leftrightarrow \#aRb > (\#aRb + \#bRa)/2 \qquad (. 11)$$

where #xRy is the number of relations R which contain (x,y).

The group preference matrix of subset1 based on the simple majority rule is given in Table 7.3. In total 35 correct responses have been used.

| nr | 12 | 13 | 15 | 18 | 20 | 27 | 34 |
|---|---|---|---|---|---|---|---|
| 12 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 13 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 18 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 20 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 27 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 34 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

*Table 7.3 Group preference (A,M) for 7 type expressions of dataset1 (k=5)*

A group ranking can be obtained from the group preference structure if the data are consistent: there are no intransitivity's (i.e. a preference cycle: $p_{ab} = 1$

$\wedge$ $p_{bc} = 1 \wedge p_{ca} = 1$) allowed. For this subset there are inconsistencies in the group preference structure. The three type expressions that are not transitive are: 12: (num $\rightarrow$ char $\rightarrow$ char); 13: [(num,bool)]; and 27: (num,bool) $\rightarrow$ (num,bool).

The group preference structure of the second subset is consistent. It is a strict weak order (asymmetric and negatively transitive). An ordinal function m for this subset is defined as follows (Roberts, 1979: 105):

$$m(x) = \#\{y \in A \text{ such that } xRy\} \qquad (. 12)$$

The function m for the subset of type expressions of dataset2 is given in Table 7.4.

| nr | function | | |
|---|---|---|---|
| 123 | m ([char] $\rightarrow$ bool) | = | 6 |
| 124 | m (bool $\rightarrow$ [char]) | = | 5 |
| 129 | m ([char]) | = | 4 |
| 132 | m (bool $\rightarrow$ char) | = | 3 |
| 127 | m (char $\rightarrow$ bool $\rightarrow$ bool) | = | 2 |
| 115 | m (bool) | = | 1 |
| 118 | m (char) | = | 0 |

*Table 7.4 Function m for type expressions in subset2*

For the first subset, this function yields the same value for each of the type expressions 12, 13 and 27, resulting in a violation of the representation condition (equation 1).

### 7.4.2.2 Probabilistic axiomatic analysis

A major disadvantage of the analysis in the previous section is that only complete preference structures can be taken into account. With a probabilistic analysis this can be circumvented. It is possible to calculate the *probability matrix* (Roberts, 1979: 273) with relative frequencies based on all correctly answered questions:

$$p_{ab} = (\#aRb) / (\#aRb + \# \neg(aRb)), \text{ if } a \neq b \qquad (. 13)$$
$$p_{ab} = 0.5, \text{ if } a = b \qquad (. 14)$$

From this it can be seen that: $\forall$ a,b $\in$ A: $p_{ab} + p_{ba} = 1$. Such a probability matrix represents a *forced choice pair comparison structure* (A,p).

This structure (A,p) is *weak stochastic transitive* if, $\forall$a,b,c $\in$ A :

$$p_{ab} \geq 0.5 \wedge p_{bc} \geq 0.5 \Rightarrow p_{ac} \geq 0.5 \qquad (. 15)$$

A weak order (A,W), associated with a weak stochastic transitive structure (A,p), is given by W defined on A by

$$aWb \Leftrightarrow p_{ab} \geq p_{ba} \qquad (. 16)$$

| nr | 12 | 13 | 15 | 18 | 20 | 27 | 34 |
|---|---|---|---|---|---|---|---|
| 12 | 0.50 | 0.67 | 1.0 | 0.71 | 0.40 | 0.43 | 0.71 |
| 13 | 0.33 | 0.50 | 0.60 | 0.58 | 0.33 | 0.44 | 0.64 |
| 15 | 0.0 | 0.40 | 0.50 | 0.25 | 0.0 | 0.0 | 0.42 |
| 18 | 0.29 | 0.42 | 0.75 | 0.50 | 0.33 | 0.30 | 0.54 |
| 20 | 0.60 | 0.67 | 1.0 | 0.67 | 0.50 | 0.60 | 0.83 |
| 27 | 0.57 | 0.56 | 1.0 | 0.70 | 0.40 | 0.50 | 0.89 |
| 34 | 0.29 | 0.36 | 0.58 | 0.46 | 0.17 | 0.11 | 0.50 |

*Table 7.5 Probability matrix for 7 type expressions of dataset1 (k=14)*

As an example, in Table 7.5 the probability matrix is given for the same subset of type expressions as in the previous section. The matrix can be compared with the group preference matrix of Table 7.3. However, the matrix presented here has been calculated with data of all 14 subjects. In total 74 correct responses have been used. This probability structure is weak stochastic transitive and hence consistent, contrary to the group preference of 5 subjects.

| rank | type expressions |
|---|---|
| 7 | 20: bool → char → bool |
| 6 | 27: (num,bool)→(num,bool) |
| 5 | 12: num → char → char |
| 4 | 13: [(num,bool)] |
| 3 | 18: bool → num |
| 2 | 34: (num,bool,char) |
| 1 | 15: num → bool |

*Table 7.6 Ranking of 7 type expressions based on associated weak order (k=14)*

On the basis of this probability structure for these type expressions, an associated weak order can be calculated with a ranking (see Table 7.6).

In the previous analysis, no attention has been given to *measurement errors* and the significance of the experimental data. For the probability matrix from

this data set (Table 7.5), the significance of the relative frequencies has been calculated. The sign test has been used[24] (Guilford & Fruchter, 1978). A significance of $\alpha < .09$ will be achieved if 10 out of 14 subjects show the same sign of the difference between the time measured for two type expressions $t_a$ and $t_b$, which presumes a probability $p_{ab} \geq 0.71$. For the probability of the type expressions in subset1, the structure (A,W) is calculated with

$$aWb \Leftrightarrow p_{ab} \geq \lambda \qquad (. 17)$$

with threshold probability $\lambda = 0.75$. The structure obtained in this case is not a weak order, however it satisfies the axioms for a *semiorder,* which are the following (Roberts, 1979: 250):

$$\neg\, aRa \qquad (. 18)$$

$$aRb \wedge cRd \Rightarrow (aRd \vee cRb) \qquad (. 19)$$

$$aRb \wedge bRc \Rightarrow (aRd \vee dRc) \qquad (. 20)$$

A weak order (A,W) associated with the semiorder (A,R) can be obtained with W defined on A by (Roberts, 1979: 256):

$$aWb \Leftrightarrow \forall\ c \in A:\ (bRc \Rightarrow aRc) \wedge (cRa \Rightarrow cRb) \qquad (. 21)$$

For the semiorder obtained above, the associated weak order has been calculated. A ranking for this weak order is given in Table 7.7, with ties at ranks 4-5 and 6-7 (resulting respectively in rank 4.5 and 6.5).

| rank | type expressions |
|---|---|
| 6.5 | 20: bool → char → bool<br>27: (num, bool) → (num, bool) |
| 4.5 | 12: num → char → char<br>18: bool → num |
| 3 | 13: [(num, bool)] |
| 2 | 34: (num, bool, char) |
| 1 | 15: num → bool |

*Table 7.7 Ranking of a subset of 7 type expressions based on the associated weak order of the semiorder (λ=0.75, k=14)*

[24] The Wilcoxon signed ranks test is not applicable because the rankings are not complete for all subjects.

From the previous analysis of the empirical order of type expressions with respect to the external attribute comprehensibility, it can be concluded that, for subsets of type expressions, the measurement of time to find an instance of a given type, results in an ordinal scale.

## 7.5 Discussion

It has been shown that type expressions can be measured on an ordinal scale with respect to the internal attribute structure by defining an extension of a containment relation on abstract type expressions.

In the case study, the comprehensibility of simple type expressions has been operationalized as a time measurement. The ranking of the average time is in reasonable agreement with a weak order extension of the partial order obtained for the corresponding abstract type expressions. Axiomatic analysis has been used to localise inconsistencies in the experimental data: an example has been given of an intransitive group preference. An ordinal measure has been calculated for a consistent data set. Incomplete data sets have been analysed with a probabilistic consistency axiom: the weak stochastic transitivity. An ordinal measure has been established based on these probabilistic data. Measurement errors have been treated with a threshold probability and semiorders. The order obtained in this way shows a deviation of the previous order and appears to have more ties.

Subsequently, the correspondence between the two measurements can be established now. There are two steps which have been described in a previous study (van den Berg *et al.,* 1993)[25]. Firstly, the structure metric function m defined in section 7.3.3 is calibrated, resulting in values for $c_i$. This can be done with standard linear regression techniques. Secondly, this calibrated function is used in the prediction of the comprehensibility values. The forecasting efficiency of the prediction has been established.

Another important aspect is the use of the approach, outlined in this chapter, to other software entities with other attributes. There seems to be at least one important field where this approach could be successful. This is the domain of complexity measures based on flowgraph modelling. An ordering of flowgraphs is given by Bache (see Fenton, 1991). A containment based order has been defined by Melton (Melton *et al.*, 1990; Fenton, 1992), and a formal axiomatic validation is presented by Zuse (1992). An experimental axiomatic testing could be carried out along the framework described in this chapter, e.g. for maintainability and structural properties.

---

[25] Chapter 6 of this thesis

The main point presented in this chapter is the role of representation axioms in the diagnostic testing (Luce, 1990) of the order of attributes of software entities. Inconsistencies can be localised. They may hint at anomalies in the experiments or weaknesses in the theory: they can be used in the development of the conceptual domain, e.g. in the choice of alternative abstractions. It has been shown that axiomatic testing may well contribute to the validation of software metrics, both formally and empirically.

# Chapter 8

## 8. Validation in the Software Metric Development Process [26]

*In this chapter the validation of software metrics will be examined. Two approaches will be combined: representational measurement theory and a validation network scheme. The development process of a software metric will be described, together with validities for the three phases of the metric development process. Representation axioms from measurement theory are used both for the formal and empirical validation. The differentiation of validities according to these phases unifies several validation approaches found in the software metric's literature.*

### 8.1 Introduction

As can be concluded from the plethora of software metrics, it is rather easy to conceive some software metric and to obtain numbers with such a metric, for example in the field of complexity measures. However, it is less clear that all these metrics are really good measures. To establish the quality of measures they have to be validated. It has been remarked that there are as many metrics as there are computer scientists[27]. A paraphrase of this statement is that there are as many types of validation as there are software metrics. Validation is defined as assessing the extent to which a measure really measures what it purports to measure (Fenton, 1991). However, this is a rather tautological formulation (Berka, 1983), and validation has to be operationalized in practice.

[26] This chapter is a shortened version of: K.G.van den Berg & P.M. van den Broek (1995), Axiomatic Validation in the Software Metric Development Process, in: A.Melton (Ed.), Software Measurement: Understanding Software Engineering, London: Thomson, Chapter 10.

[27] Ascribed to S.D.Conte

In this chapter, the development of a software metric will be traced with an explication of different aspects of validation. A simplified framework for software measurement will be used (see Figure 8.1).

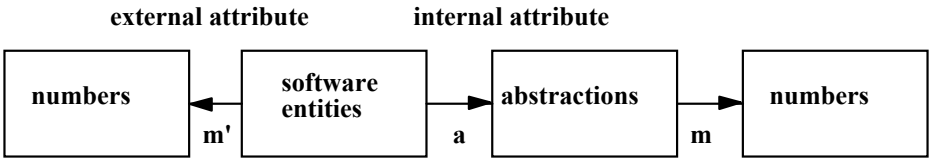



*Figure 8.1 Framework for software measurement*

Software entities will be considered: products, processes or resources (Bush & Fenton, 1990). Data on an external attribute (e.g., maintainability, reusability) of these entities are collected with some measure *m'*, the quantified criterion (Melton *et al.*, 1990). This external attribute will be related with some internal attributes, such as size or structure. The internal attribute is measured with a metric function *m* on abstractions of the software entities.

Kaposi (1990) has given an account of the role of measurement theory in software engineering. Five parts in the planning of measurement are distinguished: 1. The problem definition: designating the target objects and key properties that must be measured. 2. The modelling: a model description of the target set with reference to the key properties. 3. The forming of the empirical relational system: describing the model by means of an observable relation between the objects in terms of the selected key properties. 4. The definition of the formal relational system: selecting the system in which the measured results are to be represented. 5. The validation of the results of the measurements.

In this chapter, two approaches will be brought together: the validity network scheme, which resembles Kaposi's analysis, and the representational measurement theory. In the validity network scheme, aspects of validity are differentiated for subsequent phases of the research process. In a case study, axioms from the measurement theory will be validated, both formally and empirically, according to this scheme. The case itself is of interest to researchers in the field of programming methodology (van den Berg *et al.*, 1993). More general, the case is used to exemplify the application of representational measurement theory and aspects of validation in software measurement.

As remainder of this section, the representational measurement will be introduced briefly (section 8.1.1), followed by the validity network scheme (section 8.1.2) and the case study (section 8.1.3).

### 8.1.1 The representational measurement theory

The representational approach has been used in software measurement (e.g., Baker *et al.*, 1990; Fenton, 1991; Melton, 1990; Bieman *et al.*, 1992; Melton, 1992; Zuse, 1992). Some basic concepts in this measurement theory (Krantz *et al.*, 1971; Finkelstein & Leaning, 1984; Suppes *et al.*, 1989; Luce *et al.*, 1990) will be defined according to Roberts (1979). A pivotal concept is the order preserving mapping between relational structures. A short introduction to the representational measurement theory has been given in Chapter 7.1.

### 8.1.2 The validity network scheme

According to Brinberg and McGrath (1985), three *domains* can be distinguished in the research process: the *substantive* domain, the *conceptual* domain and the *methodological* domain. Each domain is defined by its elements, the relations between the elements, and the embedding system. The embedding system refers to the set of assumptions within which these elements and relations are studied.

In software measurement, the substantive domain consists of the empirical relational structure in the framework (Figure 8.1), together with the embedding system: the actual *context* of the software entities (e.g., industry, training). The conceptual domain consists of the other relational structures in the framework. The embedding system in the conceptual domain is called the *paradigm*. For example, structure metrics are based on the assumption of the compositionality of the structural properties. The methodological domain is primarily concerned with the mapping of the empirical relational structure into a numerical relational structure. The embedding system in this domain is the research *strategy*, for example the use of field studies or controlled experiments.

The research process itself consists of three *phases*: the *generative* or prestudy phase, the *executive* or central phase, and the *interpretative* or generalisation phase (Table 8.8).

| domain / phase | substantive | conceptual | methodological |
|---|---|---|---|
| generative | valuation validities | | |
| executive | correspondence validities | | |
| interpretative | generalisation validities | | |

*Table 8.8 The validity network scheme*

In each domain of each phase, there are specific aspects of validity depicted in the validity network scheme. In the generative phase, the *valuation* validities are of primary concern, in the executive phase the *correspondence* validities, and in the interpretative phase the *generalisation* validities. A description of the various aspects of these validities will be presented in the elaboration of the case study.

### 8.1.3 The case study [28]

In order to make the discussion of the different aspects of validation concrete, a case study will be presented related to a specific kind of software entities: software documentation. Proper documentation presumably has an impact on important quality aspects, such as maintainability and reusability. There is an interest in objective data on the impact of documentation. For the program code, the documentation problem is obvious. Besides documentation in natural language, there is a tendency to formalise documentation. On procedural level this can be done with for example preconditions and postconditions. Another possibility is the use of explicit typing by the programmer. In this case, the programmer provides information about the type of the objects in the program. (This is opposed to implicit typing, where the computer carries out the check of types as can be derived from the code.) This form of documentation not only may have an impact on the reliability of the software, but also on the comprehensibility to human readers of the programs (reviewers, maintenance programmers). In the case study, documentation in the form of explicit typing will be considered. The software entities are type expressions in the functional programming language Miranda[29]. Type expressions themselves have a certain degree of (cognitive) complexity: they are easy or difficult to comprehend. The comprehensibility will be taken as the external attribute. The internal attribute is the structure of type expressions. The relationship between the comprehensibility of type expressions and their structural properties will be investigated.

### 8.1.4 Overview

This chapter is organised as follows. First, the generative phase of a software metric will be described (section 8.2). More details about the software entities in the case study, the type expressions, will be given. Furthermore, the modelling of the structure of type expressions and the measurement of the compre-

[28] This is the same case study as described in Chapter 7 of this thesis

[29] Miranda is a trademark of Research Software Ltd.

hensibility will be elaborated on. The subsequent section 8.3 will deal with the executive phase in the development of the software metric. The actual collection of data on the external attribute, the comprehensibility, will be described. The deterministic and probabilistic testing of axioms will be exemplified. The structure metric function will be calibrated and used for prediction of the comprehensibility. The interpretative phase in the following section 8.4 will elaborate on the generalisation of the results obtained in the foregoing phases. The final section 8.5 discusses the relation of axiomatic validation presented in this chapter to other validation approaches.

## 8.2 The generative phase

Consecutively, the substantive domain, the conceptual domain and the methodological domain in the generative phase will be described. For each domain, the elements, the relations and the embedding system will be given. This section will be concluded with a discussion of the valuation validities in this phase.

### 8.2.1 The substantive domain

An outline of the substantive domain implies the *phenomena*, the *observed patterns* and the *context* in 'the real world'. The phenomenon to be studied is the comprehensibility of type expressions, which has been introduced in Chapter 7.2. For example, the type of a function *split* is:

```
split :: (num → bool) → [num] → ([num],[num])
```

Several observations have been made with respect to the role of explicit types in programming, e.g.

> *Miranda scripts often contain type declarations as these are useful for documentation and provide an extra check, since the type checker will complain if the declared type is inconsistent with the inferred one. (Turner, 1986)*

> *Types impose constraints that help to enforce correctness. Typing enforces a programming discipline on the programmer that makes programs more structured and easier to read. (Cardelli & Wegner, 1985)*

> *Judicious placement of type signatures is a good idea, since it improves readability and helps bring programming errors to light. (Hudak & Fasel, 1992)*

*Typing would make the programmer think about what kind of parameters a function will be used for and, also, would provide more information about how the program worked to anyone reading or maintaining it at a later stage. (Kosky, 1988)*

Type declarations form an important clue to the understanding of functions in a program. They give a partial specification of the function: the type of its arguments and the type of the result. The complexity of the type declaration might give an indication of the complexity of the task to be performed by the function.

In the 'real world model' (Maki & Thompson, 1973) restrictions will be imposed on the 'real world' entities and phenomena. In the case study the type expressions will be restricted to so called *simple* type expressions (no type variables, no type synonyms, no abstract data types: see Chapter 7.2).

Type expressions are studied in the context of programs developed in an academic environment. It is evident that comprehensibility depends on the experience of the reader. The case study is carried out with novice Miranda programmers with corresponding proficiency. Only structural properties of simple type expressions in Miranda in relation with their comprehensibility to novice programmers are examined.

### 8.2.2 The conceptual domain

The second domain, the conceptual domain, implies the *concepts*, the *relations*, and the *conceptual paradigm*. The concepts will be given in a relational structure with relations on abstract type expressions. The conceptual paradigm is the representational measurement theory as described above, and the compositionality of the structural properties, as expressed in structure metrics (Fenton & Kaposi, 1989).

#### 8.2.2.1 The abstraction

In the conceptual domain a relational structure $(A, R_1,...,R_n)$ is defined. Set A consists of abstract type expressions; $R_1,...,R_n$ are relations on abstract type expressions. In some cases, the corresponding operation of a relation will be used in the relational structure (cf Roberts, 1979: 41). These operations are called *concatenation operators* or *constructors*.

The mapping of simple type expressions to abstract type expressions is described in Chapter 7.3.1. For example, the abstraction of the type of the function *split* is:

```
F [ F [N,B], L N , T {L N, L N}]
```

with respectively: L the list type constructor; F the function type constructor; T the tuple type constructor; C the standard type *char* (not used is this example); N for *num* and B for *bool*. Next to the constructors, [...] denotes an ordered list of abstract type expressions, and {...} denotes a multiset.

There are alternative abstractions discussed in van den Berg *et al.* (1993). The choice between abstractions of entities is determined by the actual use of the abstractions: the establishment of a good correspondence between an internal attribute based on these abstractions, and an external attribute of the entities.

#### 8.2.2.2 The containment relation and the metric function

The containment relation on abstract type expressions, denoted by $\prec$, has been defined in Chapter 7.3.2. The containment relation $\prec$ on type expressions is a partial order.

For abstract type expressions, a linear *structure metric* function *m* is defined in Chapter 7.3.3 :

$$
\begin{array}{lcl}
m(C) & = & c_C \\
m(N) & = & c_N \\
m(B) & = & c_B \\
m(T\{t_1,\ldots,t_n\}) & = & c_T + m(t_1) + \ldots + m(t_n) \\
m(L\ t) & = & c_L + m(t) \\
m(F[t_1,\ldots,t_n]) & = & c_F + m(t_1) + \ldots + m(t_n)
\end{array}
$$

With this function *m*, a new relation $\prec_m$ on type expressions is defined as follows:

$$t_a \prec_m t_b \quad \Leftrightarrow \quad m(t_a) \leq m(t_b)$$

The relation $\prec_m$ is an *extension* of the containment relation. From a measurement theorem it has been shown that ((*texp* ,$\prec_m$), (*Re,* $\leq$), *m*) is an ordinal scale.

An abstraction of type expressions and a containment relation on abstract type expressions have been defined. An extension of this relation derived from a structure metric function provides measurement of the internal attribute structure of type expressions on an ordinal scale. This allows the investigation of a correspondence of the extension with the empirical order as given by the quantified criterion, which also maps on (*Re,*$\leq$). In the following section, the measurement of the external attribute, i.e. the comprehensibility of type expressions, will be discussed.

### 8.2.3 The methodological domain

In this section, the methodological domain - which comprises the *measures*, *comparison techniques* and the *research strategy* - is described. The collection of data on the external attribute of the software entities will be addressed. The external attribute has to be operationalized by some measure.

There are several approaches to the measurement of comprehensibility of programs. In the case study, one measure has been chosen for the comprehensibility of type expressions (van den Berg *et al.*, 1993): the time in seconds needed for a subject to read a given type expression and to conceive and typewrite an instance of an object with exactly this type in the 'standard' programming environment. The time between showing the type expression on screen and the completion of the answer is measured automatically. Afterwards, with the type checker of the programming system, the answer is marked as correct or incorrect.

The strategy for data collection is that of controlled experiments, as opposed to for example field studies. Controlled experiments have been chosen to have a better control over the instances of type expressions, and to have better control over the conditions under which the comprehensibility is measured. If a vector of measures is used, one has to compare the relative merit of each measure. This is not carried out in this exploratory study.

### 8.2.4 Validities in the generative phase

The validities in the generative phase are *valuation* validities: establishing the 'value' of elements, relations and embedding systems in each domain. In all domains there are validation criteria, which may be mutually conflicting. They are all desirable, but they cannot be maximised at the same time (Brinberg & McGrath, 1985).

*Valuation Validition in the Substantive Domain*

Three general criteria for values in the substantive domain are: the *effectiveness*, the *cost* and the *quality*. The validity of the chosen phenomena and patterns have to be considered: e.g., the value of documentation in software development; the value of type declarations in software documentation; the value of comprehensibility of type expressions. Furthermore: what is the expected improvement of the software quality by the use of good documentation; what is the cost of good documentation; what is the value and the cost of quantitative assessment of documentation quality. Finally, what is the 'value' of the chosen context with restrictions on the real world to obtain the real world model. On one hand there are restrictions on the documentation: software documentation

- formal documentation - type declarations - simple type expressions; on the other hand on the programmers: academia - novice programmers.

*Valuation Validition in the Conceptual Domain*

The three criteria in the conceptual domain are: *parsimony*, the use of fewer concepts and fewer relations in the interpretation of the problem; *scope*, the range of the problem being covered by the concepts (content validity); and *differentiation*, the amount of detail of the problem that can be interpreted with the concepts (construct validity). A prerequisite value is the consistency of the concepts and relations.

Even in a small scale case study as presented in this chapter, there are many concepts introduced and used: from the representational measurement theory, from programming theory to describe type expressions and the abstraction of type expression with the containment relation and the metric function. A size metric instead of the structure metric would probably require fewer concepts. The formal validation comprises the check on consistency of the concepts used. The scope and differentiation of the concepts are apparent in the given mapping rules from the real world model to the abstractions.

*Valuation Validition in the Methodological Domain*

In the methodological domain, the three mutually conflicting criteria are: *precision*, i.e. the accuracy of the measurement and the amount of control of the variables; *realism* of the context in which the information is obtained in relation to which that information is intended to apply or to be used; and *generalisability* with respect to the chosen entities and attributes in the problem. The chosen research strategy has to result in reliable data on the phenomena. In this exploratory study, controlled experiments have been chosen. If data are to be applied, e.g. in metric tools in industrial practice, field studies will be required. The value of a measure has to be established by comparison with other measures (the criterion validity). Only one measure for the comprehensibility has been used in the case study. The realism of the context in this study can be traced back from the given abstractions and restrictions on the real world.

The analysis of data will be derived from the axioms that have been stated in the conceptual domain in the previous section. In the executive phase, it has to be established whether or not comprehensibility of type expressions can be described in a consistent relational structure, in order to resolve the scale of measurement and to establish the correspondence with relations in the conceptual domain.

## 8.3 The executive phase

The first step in this phase is the collection of quantitative data for the observed phenomena as described in the previous section. The measure will be used as a criterion for the empirical relation between the entities in the given context. Subsequently, the data obtained with this measure will be analysed and the correspondence will be established with the relations in the conceptual domain. The section will be concluded will a discussion of the correspondence validities in this phase.

The experiments and the empirical order have been described in Chapter 7.4. The following approaches in the analysis of the data have been used. Firstly, a global analysis has been given based on the average time measured for each type expression (Chapter 7.4.1). Secondly, an axiomatic analysis of the preference of each subject between pairs of type expressions has been described with a test on intransitive group preferences (Chapter 7.4.2.1). Finally, an axiomatic analysis based on the relative frequencies of these preferences has been considered with a test on stochastic transitivity (Chapter 7.4.2.2). In the same section, measurement errors have been treated with a threshold probability and semiorders.

From this analysis of the empirical order of type expressions with respect to the external attribute comprehensibility, it can be concluded that -- for subsets of type expressions -- the measurement of time to find an instance of a given type, results in an ordinal scale.

### 8.3.1 Calibration

For each of the type expressions in the data set, an expression can be derived from the metric function *m*, as defined in section 8.2.2.2. For example, the metric value for the abstraction of the type expression of the function *split* yields the expression

$$1 \times c_T + 2 \times c_F + 3 \times c_L + 4 \times c_N + 1 \times c_B + 0 \times c_C$$

This expression can be equated to the average measured time for the correct responses. With linear regression analysis of these equations for each type expression in the experiment, the calibration of the constants $c_T$, $c_F$, $c_L$, $c_N$, $c_B$, $c_C$ has been obtained (cf. Chapter 6.4.3).

### 8.3.2 Prediction

A second data set has been obtained with subjects different from the first set, and with different type expressions. The calibrated metric function of the pre-

vious section is used to calculate the time for each type expression. The Pearson product-moment correlation coefficient (Guilford & Fruchter, 1978) between the measured values and the calculated values is 0.80. The related forecasting efficiency is 40%; i.e. a reduction in variance of the predicted comprehensibility is achieved by using the calculated metric value. It is also possible to compare the ranks of the measured and calculated values. The Spearman rank correlation coefficient is 0.74. From these results it can be concluded that there is a reasonable good agreement between the measured and predicted values in the experiment (cf. Chapter 6.4.3).

### 8.3.3 Discussion

The comprehensibility of simple type expressions has been operationalized as a time measurement. The ranking of the average time is in agreement with a simple extension of the partial order obtained for the corresponding abstract type expressions, despite a rather large standard deviation in the measured values, as has been described in Chapter 7.4. Axiomatic analysis has been used to localise inconsistencies in the experimental data: e.g. intransitive group preference. An ordinal measure has been calculated for a consistent data set. Incomplete data sets have been analysed with a probabilistic consistency axiom: the weak stochastic transitivity. An ordinal measure has been established based on these probabilistic data. Measurement errors have been treated with a threshold probability and semiorders. The order obtained in this way shows a deviation of the previous order and appears to have more ties. Calibration of the metric function, as defined for abstract type expressions, has been carried out with standard regression analysis. The prediction of the comprehensibility with this calibrated metric function shows a good agreement with the measured values from an independent data set.

### 8.3.4 Validities in the executive phase

In the executive phase, the validity of the *correspondence* between the different relational structures has to be established: the correspondence validities. Moreover, the experimental design used in the collection of data has to be validated (design validity). The main emphasis of the case study has been on the correspondence between the relational structures as given in Figure 8.1. The correspondence between the empirical relational structure with the comprehensibility of type expressions and the numerical relational structure of the time measurement has been established. This correspondence has been validated by examining the representation axioms from measurement theory. Furthermore, the correspondence between the two numerical relational structures

has been validated by the calibration and prediction with the metric function. From this analysis based on standard statistical techniques, it has been concluded that there is a good correspondence between the empirical relational structure and the formal relational structure with abstract type expressions and the containment relation. The correspondence of real world software documentation with the comprehensibility of simple type expressions has not been validated in the case study. This will be considered in the following phase.

## 8.4 The interpretative phase

In this follow-up phase, the third phase in the development process, the set of findings obtained in the executive phase are interpreted. Furthermore, the repeatability of the findings, the range of variation of elements and relations (from each of the domains) over which the set of findings holds, and boundaries beyond the set of findings do not hold, are explored.

The analysis in this case study has been restricted in many ways: the subjects in the experiment (novice programmers), and the type expressions (no grouping brackets, no type variables, no type synonyms). Furthermore, alternatives of the abstraction function are not considered. Moreover, only one comprehensibility measure has been used. This leads to questions of generalisation validities.

### 8.4.1 Validities in the interpretative phase

The validities in the interpretative phase are *generalisation* or robustness validities. For each domain this validity is the extent to which the scope and limits of a set of empirical findings can be specified with respect to the elements and relations in that domain. Generalisation validities for each domain addresses the following aspects. *Replication*: would the same set of findings occur if the study is repeated with the same set of elements and relations? *Convergence*: would the same set of findings occur if certain facets of elements and relations are varied systematically? *Differentiation* or boundary search: if a different set of findings occurs with certain facets of elements and relations varied systematically, can these differences be explained with the relational system? It is not only important to look for the conditions under which the findings will fit the hypothesis, the invariance, but also try to identify and explain the conditions under which the findings disconfirm the hypothesis, the failures of invariance's.

To give some examples: If another measure for comprehensibility had been used, would the same order be found? Would a size metric instead of the struc-

ture metric yield a good correspondence with comprehensibility? Is there an influence of the recognition of the type declaration of often used standard functions? What is the influence of programming proficiency on the order of type expressions: novices versus experts?

Another important aspect is the generalisation of the approach outlined in this chapter to other software entities with other attributes. There seems to be at least one important field where this approach could be successful. This is the domain of complexity measures based on flowgraph modelling. An ordering of flowgraphs is given by Bache (see Fenton, 1991). A containment based order has been defined by Melton (Melton *et al.*, 1990; Fenton, 1992), and a formal axiomatic validation is presented by Zuse (1992). An experimental axiomatic validation could be carried out along the framework described in this chapter, e.g. for maintainability and structural properties.

There is also a questioning of the conceptual paradigm chosen in this chapter: representational measurement theory. Although this approach has a wide adherence in especially natural science, there are also meta-theoretical limitations, among others the absence of criteria to chose between alternative representations (Roberts, 1979). Another critical observation is made by Guttman:

> *There is much to be learned from exploring axioms and their formal consequences. But there remains the danger of seeking data merely to fit axioms. (Guttman, 1971; cited in Schwager, 1988).*

## 8.5 Relation with other validation approaches

There are two types of conclusions: firstly, on the topic of the case study itself, i.e. the comprehensibility of type expressions; secondly, on the validation approach as exemplified by the case study. In regard to the first point: some conclusions have been given in the discussion section of the executive phase and in the interpretative phase. As has been described in Chapter 10, the main point is the role of representation axioms in the diagnostic testing (Luce, 1990) of the empirical order of the comprehensibility of type expressions. Inconsistencies can be localised. They may hint at anomalies in the experiment or weaknesses in the theory: they can be used in the development of the conceptual domain, e.g. in the choice of alternative abstractions of type expressions.

Other approaches to the validation of software metrics can be found in the literature. Gustafson *et al.* (1992) present a classification of validation studies of software metrics. In their view, validation checks the predictive abilities of a measure against a dependent variable, while verification checks the reasonableness of the measure. For each approach it is indicated: whether verifica-

tion or validation is achieved, whether the approach produces a dependent variable, whether there is an underlying theory used or produced, and whether the results of the approach are generalizable to other data or environments. They distinguish the following (not disjunct) approaches: 1. The shotgun approach: using statistical correlation techniques between many measures. 2. The standard dependent variable approach: based on a theory with data from completed projects. 3. The controlled-experiments approach: based on a theory with data from experiments. 4. The verification approach: using formal properties of measures. 5. The exploratory approach: in which a large set of measurements is grouped with factor analysis. 6. The intuitive approach: in which measurements are correlated with judgement of experts. 7. The goal-oriented approach: in which a particular property has to be optimised. 8. De facto approaches, which fail to be classified in one of the previous categories. They conclude that the lack of planning for validation in the development of measures will result in measures that have limited usefulness and questionable validity.

Schneidewind (1992) presents a methodology for validating software metrics, from the point of view of the metric user. He discusses six validity criteria: 1. Association: the extent to which a variation in a software attribute is explained by the measure. 2. Consistency: the strength of the rank correlation between a software attribute and a measure. 3. Discriminative power: the strength of classification of a software attribute with a measure. 4. Tracking: a monotonic relation between attribute and measurement. 5. Predictability: the accuracy of predicting an attribute with a measure. 6. Repeatability: the success rate of validating the measure for an attribute. The six criteria support the three functions of measurement: assessment, control and prediction. The criteria provide a rationale for the validation, the selection and application of metrics.

As compared with these approaches, in this chapter the emphasis is on the validation of representation axioms in the different phases of the software metric development process, both formally and empirically. This approach is especially useful in a domain with a weak theoretical foundation. Validities have been differentiated in the validity network scheme. The criteria listed by Schneidewind can be found in this network. The development process is usually not a linear process, but will be iterated in a spiral development. The approaches distinguished by Gustafson *et al.* (1992) may have their own merits in different phases of this spiral process. A standard on validation issues in software measurement is urgently required (cf. American Psychological Association, 1954).

# Chapter 9

## 9. Programmers' Performance on Structured versus Nonstructured Function Definitions [30]

*A control-flow model for functional programs is used in an experimental comparison of the performance of programmers on structured versus nonstructured Miranda function definitions. The experimental set-up is similar to the Scanlan study (1989). However, in the present study, a two-factor repeated measures design is used in the statistical analysis. The control-flow model appears to be useful in the shaping of the experiment. A significantly better performance has been found for structured function definitions on both dependent variables: the time needed to answer questions about the function definitions and the proportion correct answers. Moreover, for structured function definitions, a counter-intuitive result has been obtained: there are significantly fewer errors in larger definitions than in smaller ones.*

### 9.1 Introduction

There is a long standing discussion on structured programming in the literature (e.g. the survey of Vessey & Weber, 1984). Most of this research has been carried out in the domain of imperative programming. This chapter will present an experiment on programmers' performance in the domain of functional programming for structured versus nonstructured function definitions. The set-up of this experiment is similar to the Scanlan study (1989) in his comparison of structured flowcharts and pseudocode. However, our experimental design and statistical analysis are different from this study: these differences will be explained in the subsequent sections. The characterisation of the structure of function definitions is based on a control-flow model as defined in a

[30] K.G. van den Berg & P.M. van den Broek (1996), Programmers' Performance on Structured versus Nonstructured Function Definitions, *Information and Software Technology 38(7) pp 477-492*

previous paper (van den Berg & van den Broek, 1995a) (Chapter 5 of this thesis). The framework for experimentation in software engineering (Basili *et al.*, 1986) will be used in the following outline of the present study.

The *motivation* of this study has been indicated above: an experimental comparison of programmers' performance on structured versus nonstructured 'programs' in the domain of functional programming. The actual *objects* in this study are Miranda function definitions (Turner, 1986). The attribute structure of function definitions will be defined in terms of a control-flow model (van den Berg *et al.,* 1995a). The control-flow in function definitions is determined by patterns and guards, as will be described in sections 9.2 and 9.3.

The *purpose* of this study is to validate empirically some programming style rules on the use of guards and patterns in function definitions with respect to programmers' performance.

> *Pattern matching is one of the cornerstones of an equational style of definition; more often than not it leads to a cleaner and more readily understandable definition than a style based on conditional equations [with guards]. (Bird & Wadler, 1988)*

One *perspective* in this study on programmers' performance is that of a formal technical review of coding or a code walkthrough: i.e. inspection of code written by another programmer (Pressman, 1994). Programmers have to comprehend the code and make statements about the behaviour of the program. The *domain* of this study can be characterised as programming-in-the-small by novice programmers (Computer Science students). The *scope* of the study is that of a single programmer working on a single program-unit (a Miranda function definition).

In the following section, patterns and guards in function definitions are described in more detail, and in section 9.3 the control-flow model is recapitulated. In sections 9.4 - 9.7 the experiment will be described, with the results in section 9.8, followed by a discussion and some conclusions.

## 9.2 Function definitions

A description of patterns and guards in Miranda function definitions is given in Peyton Jones (1987). An example is given in Table 9.1: a definition of the function *split* (the line numbers have been added). The function *split* returns, for given a predicate, i.e. a boolean function with type (* → bool), and a list with type [*], a tuple with two components: the first component is the list with elements satisfying the predicate and the second component is the list with elements not satisfying the predicate. In line 1, the type of the function *split* is

given: it is a polymorphic function (a star * denotes a type variable). For example, evaluation of the expression *split even* [*2,4,7,4*] yields the tuple *([2,4,4],[7])*.

```
split :: (* -> bool) -> [*] -> ([*],[*])                    1
split p []                                                   2
  = ([],[])                                                  3
split p (x:xs)                                               4
  = (x : ys, zs ), if p x                                    5
  = (ys, x : zs ), if ~ (p x)                                6
     where (ys,zs) = split p xs                              7
```

*Table 9.1 Definition of the function split*

The second argument of the function *split* is a list. In the first clause of the definition (line 2-3), the pattern [ ] for an empty list is used for the selection of this clause. In the second clause (line 4-7), the non-empty list pattern *(x:xs)* is used. In the second clause there are two cases, one with the guard *p x* (line 5), the other with the guard *~ (p x)*. In the local definition on line 7, the tuple *(xs,ys)* is defined in terms of a recursive call of *split*.

The patterns in this definition are *disjoint*: if one pattern matches, then there is no other pattern that will match. E.g., if the actual argument list matches the pattern [ ] then no other pattern will match, and the same applies to pattern *(x:xs)*. Moreover, these patterns are *exhaustive*: for any argument there will be a pattern that will match. E.g., a list-argument is either empty and matches the pattern [ ], or it is non-empty and matches the pattern *(x:xs)*. The guards in this definition are disjoint as well: if *p x* is True then no other guard is True; moreover, these guards are exhaustive: either *p x* is True or *~(p x)* is True.

In this definition of *split*, the meaning of the definition is independent of the *textual order* of the clauses and cases. However, quite commonly, the meaning depends on the order of the clauses and the cases. Moreover, guards in Miranda function definitions may interact with pattern matching. There are few examples in the literature to demonstrate the latter.

In the first example, the function *funnyLastElt* returns the first negative element of its argument list, or if there is no such element, it returns the last element of this list (Peyton Jones, 1987) (see Table 9.2). If an argument list is not empty, then the first clause is selected and the guard *x<0* will be evaluated. If this condition is True, the function returns *x*; otherwise (because there is no other guard), the following pattern *(x:*[ ]*)* will be checked, and so on. The

meaning of the definition depends on the order of the clauses: e.g., if one exchanges the first clause with the last, the function does not satisfy the given specification anymore.

```
funnyLastElt :: [num] -> num
funnyLastElt (x:xs) = x, if x<0
funnyLastElt (x:[]) = x
funnyLastElt (x:xs) = funnyLastElt xs
```

*Table 9.2 Definition of the function funnyLastElt*

Another example is given by Holyer (1991) with an (unusual) definition of the Miranda standard function *drop* (see Table 9.3). The function is specified as follows: *drop n xs* removes the first *n* elements from the argument list *xs;* if *n* is not an integer, there will be a program error; if *n* is negative the argument list will be returned.

```
drop :: num -> [*] -> [*]
drop n xs          = error "fractional", if ~ integer n
                   = xs, if n<=0 \/ xs=[]
drop (n+1) (x:xs) = drop n xs
```

*Table 9.3 Definition of the function drop*

In the second clause of this definition, there is a matching on the list pattern *(x:xs)* and on the integer pattern *(n+1).* The meaning of the definition depends on the order of the clauses as well as of the order of the guards.

The interaction of patterns and guards implies that there have to be rules about the operational semantics. As stated above, for Miranda these rules are: patterns in a clause are evaluated from left to right, and guards in textual order; and clauses are evaluated in textual order (Peyton Jones, 1987). Obviously, there is an operational bias in the language design of Miranda (Petre & Winder, 1990). The control-flow model (van den Berg & van den Broek, 1995a) captures the operational semantics of function definitions.

Some *programming style rules* with respect to use of patterns and guards in function definitions can be found in the literature, each with a different strength:

1. Use total function definitions (both exhaustive patterns and exhaustive guards: Plasmeijer & van Eekelen, 1994)

2. Use order independent clauses in function definition (Bird & Wadler, 1988; Peyton Jones, 1987)
3. Use exhaustive patterns (Bird & Wadler, 1988)
4. Use disjoint patterns (Holyer, 1991)
5. Use order independent alternatives for each clause (Bird & Wadler, 1988)
6. Use exhaustive guards (Plasmeijer & van Eekelen, 1994)
7. Use disjoint guards (Plasmeijer & van Eekelen, 1994)

One of such rules (see section 9.4) is the subject of experimental validation as will be described in the subsequent sections. First, the control-flow model will be recapitulated.

## 9.3 Control-flow model

Flowgraphs are used for the modelling of control-flow in imperative programs (Fenton, 1991). The nodes in the directed graphs correspond to statements in the programs, whereas the edges from one node to the other indicate a flow of control between corresponding statements. The stop node in a flowgraph has outdegree zero, and every node lies on some path from the start node to the stop node. The nodes with outdegree equal to 1 are called procedure nodes; all other nodes are termed predicate nodes. E.g., an elementary action is modelled as flowgraph $P_1$ in Figure 9.1a; the if-then construct in a program is modelled as flowgraph $D_0$ in Figure 9.1b; the if-then-else construct is modelled as flowgraph $D_1$ in Figure 9.1c.

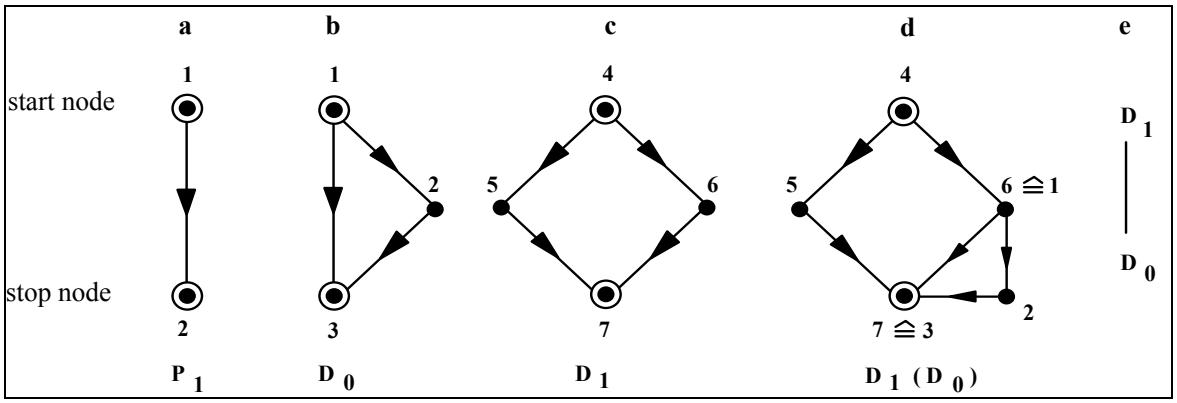



*Figure 9.1 Elementary flowgraphs and decomposition tree*

Flowgraphs can be concatenated (sequencing) to a new flowgraph; and flowgraphs can be nested on each other. An example of nesting $D_0$ onto $D_1$ at node 6 in Figure 9.1c, is given in Figure 9.1d. This is denoted as $D_1(D_0)$, in which is abstracted from the node onto which is nested. Associated with any flowgraph is a decomposition tree which describes how the flowgraph is built by sequenc-

ing and nesting elementary flowgraphs, such as $D_0$ and $D_1$. The decomposition tree of the flowgraph in Figure 9.1d is depicted in Figure 9.1e.

The operational semantics of Miranda function definitions is captured in the control-flow model (van den Berg & van den Broek, 1995a). For example, the control-flow graph for the function definition *split* is given in Figure 9.2.

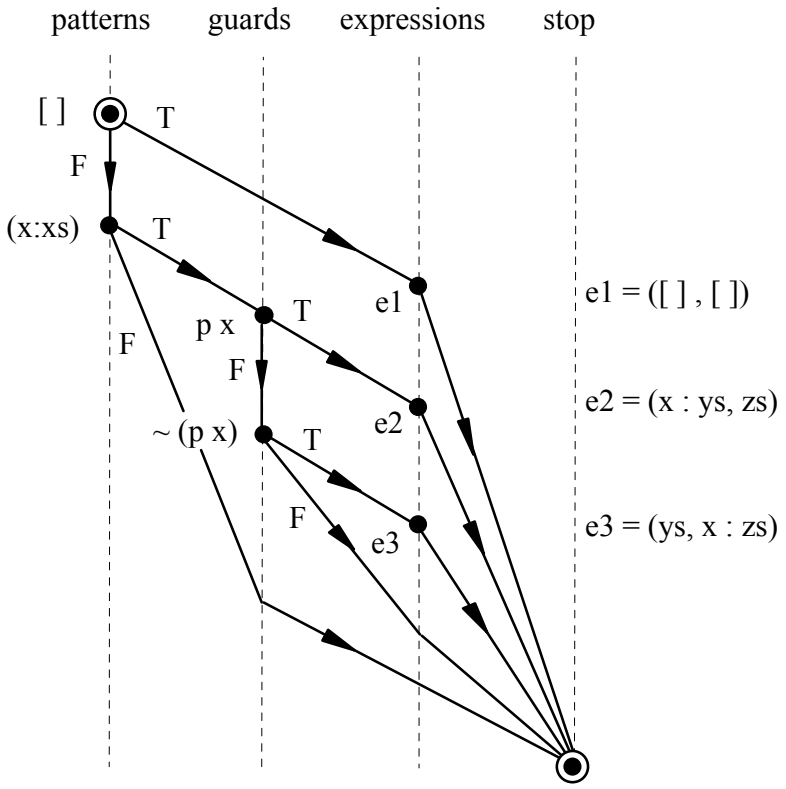



*Figure 9.2 Annotated control-flow graph of the function split*

The four vertical lines indicate the kind of nodes in these flowgraphs: predicate nodes (outdegree 2) for patterns and guards, procedure nodes (outdegree 1) for the expressions, and finally the stop node (outdegree 0). For the predicate nodes, the True (T) and False (F) branches are indicated. Note that the lower (False) branch starting at the pattern *(x:xs)* is infeasible because either the pattern [ ] or the pattern *(x:xs)* will succeed: these two patterns are exhaustive. The same applies to the lower (False) branch starting at the guard ~(p x): in any case, one of these guards will give the value True. However, in this model is abstracted from the actual content of the patterns and guards.

Flowgraphs can be uniquely decomposed into a hierarchy of (indecomposable) prime flowgraphs. E.g., the decomposition of the flowgraph given in Figure 9.2 is $D_1(D_0(D_1(D_0)))$. In this case, the depth of decomposition is 4.

We will give some additional definitions, as will be used in the description of the experiment in the following section. A *path* in a flowgraph is a sequence of consecutive nodes from the start node to the stop node. A *D-structured* path is given by a sequence of patterns followed by a sequence of guards, then possibly an expression node, and then the stop node. An *X-structured* path[31] is a path that is not D-structured: i.e., a sequence in which a guard is followed by a pattern. (In flowgraphs as drawn in Figure 9.2 and Figure 9.3, an X-structured path can be identified by an edge directed from right to left.) A D-structured path and an X-structured path are called *similar* if the D-structured path is a permutation of the X-structured path. A function definition is *structured (D-structured)* if all paths in its flowgraph are D-structured; otherwise the definition is *nonstructured* (*X-structured).*

Function definitions are called *comparable* if their flowgraphs contains the same predicate nodes (patterns, guards), and the expression nodes represent simple numeric constants. (Comparable functions need not to be semantically equivalent). Two example scripts with comparable definitions are given in Table 9.4.

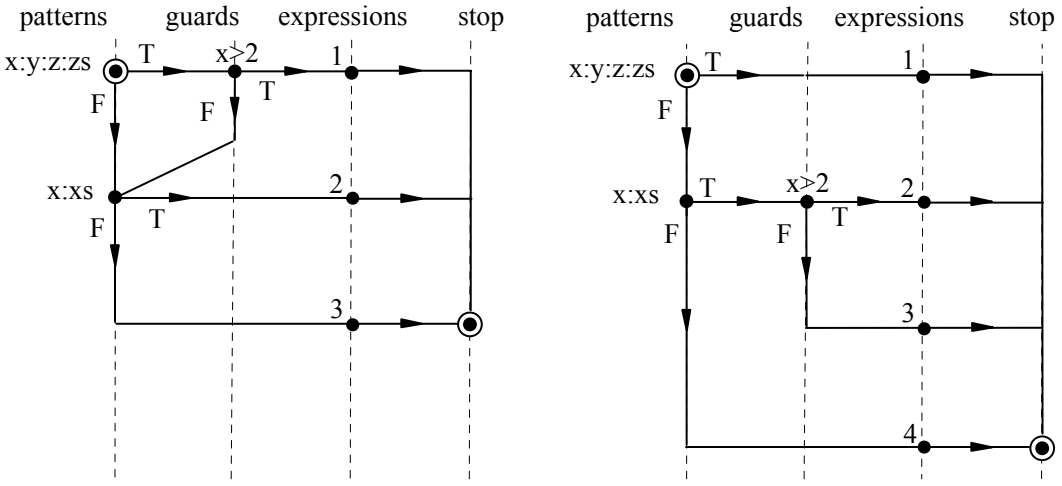
```
|| script 101
f :: [num] -> num

f (x:y:z:zs) = 1, if x>2
f (x:xs)     = 2
f xs         = 3

top = f [1,2,3]
```

```
|| script 103
f :: [num] -> num

f (x:y:z:zs) = 1
f (x:xs)     = 2, if x>2
             = 3, otherwise
f xs         = 4

top = f [3,4]
```

*Table 9.4 An X-structured function definition in script 101 and a comparable D-structured function definition in script 103*

[31] X refers to a prime other than $D_0$ and $D_1$

The flowgraphs of these function definitions[32] are given in Figure 9.3. The function *f* in script 101 is X-structured: it contains an X-structured path (with the edge from the guard x>2 to the pattern x:xs). The function *f* in script 103 is D-structured: it contains only D-structured paths.




a. flowgraph of function f in script 101

b. flowgraph of function f in script 103

*Figure 9.3 Flowgraphs of functions in script 101 and 103*

The evaluation of `top = f [1,2,3]` in script 101 results in an X-structured path with the following sequence of nodes (length of path = 6):

```
< start node,(x:y:z:zs), x>2,(x:xs), 2, stop node >
```

The evaluation of `top = f [3,4]` in script 103 results in a D-structured path:

```
< start node,(x:y:z:zs),(x:xs), x>2, 2, stop node >
```

Moreover, these two paths are similar: the path in script 101 is a permutation of the path in script 103.

The *hypothesis* is that programmers' performance on the D-structured path is better than on the similar X-structured path, and hence (this is our *assumption*) that structured function definitions are better than comparable nonstructured function definitions. In the subsequent section, criteria for programmers' performance are established, and an experiment is described to test this hypothesis.

[32] The numbers refer to the script numbers used in the experiment (see section 9.6)

## 9.4 Experiment

In this section the design of the experiment will be described. The aim of the experiment is to validate the following programming style rule: '*Use structured function definitions instead of nonstructured ones'*. In the experiment we will test the performance of programmers on structured versus comparable non-structured function definitions. The experimental set-up is similar to the one used by Scanlan (1989) in the comparison of structured flowcharts and pseudocode. However, the experimental design and the statistical analysis presented here are different. These differences will be brought up in the subsequent sections. In section 9.4.1 and 9.4.2, we will consider the independent and dependent variables in our experiment, followed by a description of the experimental design (9.4.3), the statistical model (9.4.4) and the hypotheses (9.4.5).

### 9.4.1 Independent variables

The two independent variables in the experiment are the following:

- The *Size* of a script with the function definition and the top expression. The three levels of *Size*, i.e. *Small*, *Medium* and *Large*, are characterised by the length of the path belonging to the top expression, the net lines of code of the scripts[33] (NLOC), and some control-flow metrics (van den Berg & van den Broek, 1995a) (see Table 9.5). Despite the relatively small number lines of code, the function definitions - especially the larger ones - are rather complicated (e.g., McCabe's cyclomatic complexity number and the depth of decomposition).

| Metric<br>Level | path length | NLOC | #nodes | McCabe's cycl comp | depth of decomposition |
|---|---|---|---|---|---|
| Small | 6 | 5-6 | 7-8 | 4 | 2-3 |
| Medium | 9 | 8-9 | 13-14 | 7 | 4-6 |
| Large | 12 | 13-14 | 21-22 | 11 | 6-9 |

*Table 9.5 Properties of Small, Medium and Large function definitions*

- The *Structure* of the function definition in a script. The two levels of *Structure* are the nonstructured function definition (*X-structured*) and the structured function definition (*D-structured*), as described in section 9.3.

[33] The blank lines and comment lines are not counted; the scripts also contain the type of the function and the top level expression (cf. Table 9.4)

### 9.4.2 Dependent variables

The basic condition in the experiment is: *no time pressure*, i.e. the subjects are allowed to spend as much time as they need to answer the questions (cf. Scanlan, 1989).

> *The most basis task [in computer programming], and yet in some ways the hardest to measure, is program comprehension. (Moher & Schneider, 1982)*

The dependent variables in this experiment are two criteria on programmers' performance:

- the time to answer (*Time*), i.e. the number of seconds the subjects viewed the script and spent answering the question about the script. This is a continuous random variable.
- the correctness of the answer (*Correctness*), i.e. the answer given by the subject about the script is either correct or wrong. This is a binary random variable.

In section 9.6, the questions about the scripts used in the experiment are described in more detail.

### 9.4.3 Experimental design

The experimental design will be considered as a two-factor design with two dependent variables and six treatments. A treatment corresponds to a combination of factor levels. The first factor is the *Structure* with two levels: *structured* (D) and *nonstructured* (X). The second factor is the *Size* with three levels: *Small* (S), *Medium* (M) and *Large* (L). The design has been given schematically in Table 9.6.

| Factor | | Size | | |
|---|---|---|---|---|
| | Levels | Small<br>S | Medium<br>M | Large<br>L |
| Structure | Nonstructured<br>X | treatment<br>SX | treatment<br>MX | treatment<br>LX |
| | Structured<br>D | treatment<br>SD | treatment<br>MD | treatment<br>LD |

*Table 9.6 Experimental design with factors, levels and treatments*

Each treatment in the design will be given to each subject (as in the Scanlan study, 1989). The subjects are viewed as a random sample from a population. This is a two-factor experiment with repeated measures on all treatments,

equal sample sizes, random subject effects and fixed factor effects (Neter *et al.*, 1990). (This is contrary to the one-factor repeated-measures design as conceived by Scanlan).

| Factor | | Size | | | |
|---|---|---|---|---|---|
| | Levels | Small S | Medium M | Large L | |
| Structure | Non-structured X | $\mu_{SX}$ = mean for treatment SX | $\mu_{MX}$ = mean for treatment MX | $\mu_{LX}$ = mean for treatment LX | $\mu_{.X}$ = mean for factor level X |
| | Structured D | $\mu_{SD}$ = mean for treatment SD | $\mu_{MD}$ = mean for treatment MD | $\mu_{LD}$ = mean for treatment LD | $\mu_{.D}$ = mean for factor level D |
| | | $\mu_{S.}$ = mean for factor level S | $\mu_{M.}$ = mean for factor level M | $\mu_{L.}$ = mean for factor level L | $\mu_{..}$ = overall mean |

*Table 9.7 Experimental design with treatment means and factor level means*

The number of levels for the factor *Size* is a (a=3); the number of levels for the factor *Structure* is b (b=2). The treatment mean at level j of *Size* and level k of *Structure* will be denoted by $\mu_{jk}$ with $j \in \{S, M, L\}$ and $k \in \{X, D\}$.

We will use the following point notation for the factor level means:

$\mu_{j.} = \Sigma k\ \mu_{jk} / b$ and $\mu_{.k} = \Sigma j\ \mu_{jk} / a$

The overall mean is denoted by $\mu_{..}$ with

$\mu_{..} = \Sigma j\ \Sigma k\ \mu_{jk} / ab = \Sigma k\ \mu_{j.} / a = \Sigma j\ \mu_{.k} / b$

The denotation for the treatment means and the factor level means are given in Table 9.7.

### 9.4.4 Statistical model

The following model will be used in the statistical analysis of the experimental design described in the previous section (Neter *et al.*, 1990).

Let $y_{ijk}$ be the observed value on the dependent random variable $Y_{ijk}$ for subject i ($i \in [1..n]$) for the factor A (here *Size*) at level j and the factor B (here *Structure*) at level k.

Then, we assume the following *repeated measures model*[34] with:

$Y_{ijk} = \mu_{..} + \eta_i + \alpha_j + \beta_k + \gamma_{jk} + \varepsilon_{ijk}$

In this model, we assume that:

$\mu_{..}$ is the overall effect

$\eta_i$ is the random effect of subject i

$\alpha_j$ is the fixed effect of factor A (*Size*) at level j

$\alpha_j = \mu_{j.} - \mu_{..}$ with $j \in \{S,M,L\}$

$\beta_k$ is the fixed effect of factor B (*Structure*) at level k

$\beta_k = \mu_{.k} - \mu_{..}$ with $k \in \{X,D\}$

$\gamma_{jk}$ is the interaction effect of factor A at level j and factor B at level k

$\gamma_{jk} = \mu_{jk} - \mu_{j.} - \mu_{.k} + \mu_{..}$ with $k \in \{X,D\}$ and $j \in \{S,M,L\}$

$\varepsilon_{ijk}$ is the random error effect

with:

$\mu_{..}$ is a constant

$\eta_i$ are independent and normally distributed $N(0, \sigma^2_{\eta})$

$\alpha_j$ are constants with $\Sigma_k \alpha_j = 0$ for all j

$\beta_k$ are constants with $\Sigma_j \beta_k = 0$ for all k

$\gamma_{jk}$ are constants with $\Sigma_j \gamma_{jk} = 0$ for all k and $\Sigma_k \gamma_{jk} = 0$ for all j

$\varepsilon_{ijk}$ are independent and normally distributed $N(0, \sigma^2)$

$\eta_i$ and $\varepsilon_{ijk}$ are independent

$i \in \{1,...,n\}$; $j \in \{S, M, L\}$; $k \in \{X, D\}$

The properties of $Y_{ijk}$ are the following:

the expected value $E(Y_{ijk}) = \mu_{jk} = \mu_{..} + \alpha_j + \beta_k + \gamma_{jk}$

the variance $var(Y_{ijk}) = \sigma^2_{\eta} + \sigma^2$

the covariance $cov(Y_{ijk}, Y_{ij'k'}) = \sigma^2_{\eta}$ with not both j = j' and k = k'

Thus, this repeated measures model assumes that the variable $Y_{ijk}$ have constant variance, and that any two treatment observations for the same subject in advance of the random trials have constant covariance. Any two observations from different subjects in advance of the random trials are independent. Finally, all random variables are assumed to be normally distributed. Once the subjects have been selected, repeated measures model assumes that all of the treatment observations for a given subject are independent - that is, that there are no interference effects, such as order effects or carry-over effects from one treatment to the next.

### 9.4.5 Hypotheses

The initial three hypotheses to be tested in this study are the following: whether or not there is an interaction effect of the factors *Structure* and *Size*

[34] Only the main factor effects and the interaction effect between the main factors are considered

on each of the two dependent variables (*Time* and *Correctness*), and whether or not there is a main effect of each of these factors. There is *interaction* if the effect of the factor *Structure* on a dependent variable depends on the level of the factor *Size*, and vice versa. In Table 9.8, these null hypotheses with their alternatives are stated in terms of the model; $H_0^i$ denotes the i-th null hypothesis, and $H_a^i$ denotes the corresponding i-th alternative hypothesis. The level of significance $\alpha = 0.05$.

| Factor Effects | Null Hypothesis $H_0$ | Alternative Hypothesis $H_a$ |
|---|---|---|
| *Structure* × *Size* Interaction Effects | $H_0^1$: all $\gamma_{jk} = 0$ | $H_a^1$: not all $\gamma_{jk}$ equal zero |
| *Size* Main Effects | $H_0^2$: all $\alpha_j = 0$ | $H_a^2$: not all $\alpha_j$ equal zero |
| *Structure* Main Effects | $H_0^3$: all $\beta_k = 0$ | $H_a^3$: not all $\beta_k$ equal zero |

*Table 9.8 Hypotheses on Factor Effects*

Hypotheses on nine selected pairwise comparisons of treatment means on each of the two dependent variables will be tested as well, in particular if there is interaction. The specific null hypotheses, with the alternatives, are given in Table 9.9.

| Treatment | | Null Hypothesis | Alternative Hypothesis |
|---|---|---|---|
| Structure | Size at level Small | $H_0^4 : \mu_{SX} - \mu_{SD} = 0$ | $H_a^4 : \mu_{SX} - \mu_{SD} \neq 0$ |
| | Size at level Medium | $H_0^5 : \mu_{MX} - \mu_{MD} = 0$ | $H_a^5 : \mu_{MX} - \mu_{MD} \neq 0$ |
| | Size at level Large | $H_0^6 : \mu_{LX} - \mu_{LD} = 0$ | $H_a^6 : \mu_{LX} - \mu_{LD} \neq 0$ |
| Size | Structure at level X | $H_0^7 : \mu_{MX} - \mu_{SX} = 0$<br>$H_0^8 : \mu_{LX} - \mu_{MX} = 0$<br>$H_0^9 : \mu_{LX} - \mu_{SX} = 0$ | $H_a^7 : \mu_{MX} - \mu_{SX} \neq 0$<br>$H_a^8 : \mu_{LX} - \mu_{MX} \neq 0$<br>$H_a^9 : \mu_{LX} - \mu_{SX} \neq 0$ |
| | Structure at level D | $H_0^{10} : \mu_{MD} - \mu_{SD} = 0$<br>$H_0^{11} : \mu_{LD} - \mu_{MD} = 0$<br>$H_0^{12} : \mu_{LD} - \mu_{SD} = 0$ | $H_a^{10} : \mu_{MD} - \mu_{SD} \neq 0$<br>$H_a^{11} : \mu_{LD} - \mu_{MD} \neq 0$<br>$H_a^{12} : \mu_{LD} - \mu_{SD} \neq 0$ |

*Table 9.9 Hypotheses on Treatment Means for the variable Time*

The tests are carried out on the same data set, and therefore the tests are dependent. We will set a family level of significance of $\alpha = 0.10$. The individual

significance level for each hypothesis will be derived from this value by using one of the methods for multiple comparisons (Neter *et al.*, 1990).

## 9.5 Subjects

All subjects in the experiment are first and second year students at the University of Twente in Computer Science or Business Information Technology: in total 103 students participated in the experiment. They all completed successfully at least one course on Functional Programming (Joosten *et al.*, 1994).

## 9.6 Objects

The objects in the experiments are Miranda scripts with a function definition and a top expression. For each *Size* (Small, Medium and Large), an X-version of a script and a corresponding D-version is constructed (see Table 9.10).

| Nonstructured or X-version | Structured or D-version |
|---|---|
| an X-structured function definition + a top expression with an X-structured path | a comparable D-structured function definition + a top expression with a similar D-structured path |

*Table 9.10 The X-version and D-version of scripts*

The two paths in the two versions are similar: the X-structured path is a permutation of the D-structured path. The length of the path (cf. section 9.3) is for Small scripts equal to 6; for Medium scripts: 9; and for Large scripts: 12.

For each size, an X-version of a script is set up and a comparable D-version of this script. Two sets of comparable scripts are set up: e.g., script 101 is comparable to script 102, and so on (see Table 9.11).

| Size | | Small | Medium | Large |
|---|---|---|---|---|
| Set 1 | X-version | 101 | 105 | 109 |
| | D-version | 102 | 106 | 110 |
| Set 2 | X-version | 104 | 108 | 112 |
| | D-version | 103 | 107 | 111 |

*Table 9.11 Two sets of scripts*

Two sets are used in order to reduce practice effects (cf. Scanlan, 1989). A subject tested on an X-version out of the first set will be tested on the D-version out of the second set, and vice versa (see Table 9.12).

| | Small | | Medium | | Large | |
|---|---|---|---|---|---|---|
| group 1 | 101 X | 103 D | 105 X | 107 D | 109 X | 111 D |
| group 2 | 102 D | 104 X | 106 D | 108 X | 110 D | 112 X |

*Table 9.12 Scripts (X- and D-versions) for two groups of subjects*

An example of an X-version and a comparable D-version of small scripts (scripts 101 and 103) is given in Table 9.4. Two types of questions about these scripts can be distinguished: *forward questions* (for a given input to derive the possible outputs) and *backward questions* (for a given output to derive the conditions on the input) (cf. Green, 1980). For each version an example question with the answer is given in Table 9.13.

| question | script 101: X-version | script 103: D-version |
|---|---|---|
| forward | the given input: [1,2,3]<br>the conditions are:<br>(x:y:z:zs) ∧ ¬ (x>2) ∧ (x:xs)<br>the resulting output is: 2 | the given input: [3,4]<br>the conditions are:<br>(x:y:z:zs) ∧ (x:xs) ∧ (x>2)<br>the resulting output is: 2 |
| backward | the given output: 2<br>the conditions on the input are:<br>((x:y:z:zs) ∧ (x:xs) ∧ (x>2)) ∨<br>(¬(x:y:z:zs) ∧ (x:xs)) | the given output: 2<br>the conditions on the input are:<br>¬(x:y:z:zs) ∧ (x:xs) ∧ (x>2) |

*Table 9.13 Examples of forward questions and backward questions*

In the X-structured version (script 101) in Figure 9.3 there are two paths to expression 2; in D-structured function definitions there is only one path to each expression. As can be seen in this example with forward questions, the sequence of conditions in the X-version is a permutation of the conditions in the D-version.

In this study, forward questions with simple numeric output expressions were applied to avoid problems with the skill of subjects to draw up the conditions on the input. In the experiment, the question for each script is:

*Give the value of* top *(1 or 2 or ... or 99 if* top *yields a program error).*

The actual test objects for each subject in the experiment are six Miranda scripts each with the question about the value of top.

## 9.7 Procedure

The following procedure in the experiment has been established (after a pilot study with 8 expert programmers) (cf. Scanlan, 1989):

- the subjects did the experiment in groups of about 20, each subject at his own PC (UNIX on PC's connected to SUN-workstations)
- the subjects answered the questions as an assignment in a regular laboratory session in the computer room
- the subjects have been assigned randomly to one of the groups of scripts (see Table 9.12)
- the instruction was given on screen: there is no influence of the variability of a human instructor
- four example scripts, with the question about the value of top, were offered in fixed order to each subject; the feedback on the answers is just 'correct' or 'not correct'
- after the instruction and the example questions, the six treatment scripts with questions were offered to the subjects
- for each subject, a random permutation was used of these six scripts: this is done to balance out practice and fatigue effects; no feedback is given on the answers to these questions
- all subjects serve in all treatments (D- & X-structured for small, medium and large scripts) resulting in a repeated measures design
- the collection of the data on time and correctness of the answer has been automated (resulting in a log-file for each subject and a file with data of all subjects)

## 9.8 Results

In this section, the results of the experiment will be given: the outliers in section 9.8.1, the analysis of variance for *Time* in section 9.8.3, and for *Correctness* in section 9.8.4. The experiment has been carried out with 103 subjects. The average time they spent on the whole experiment (instruction, example scripts, treatment scripts) was 10.5 minutes (standard deviation 2.5 minutes).

### 9.8.1 Outliers

For each subject there are 6 measurements on the dependent variable *Time*, i.e. for each of the treatments. For 103 subjects there are 618 time measurements. Outliers on the measurement of *Time* have been detected on the basis

of the externally studentized residual[35] (Myers & Well, 1991). If for a subject the absolute value of the residual exceeds the value 3.0 then all measurements for this subject are disregarded. There appear to be 9 outliers[36] with the residual value ≥ 3.0 for 9 different subjects. It had been noticed that some subjects were distracted by external events during the experiment in the computer room: this could be a reason for the extreme outliers. The data of these subjects is disregarded, so of the remaining 94 subjects 564 time measurements are used in the testing of the hypothesis, together with the related measurement of the correctness.

### 9.8.2 Analysis of variance

The effects of the factors *Size* and *Structure* on the dependent variables *Time* and *Correctness* have been established. The strategy for this analysis is given by (Neter *et al.*, 1990: Ch 19) using the variance of the data. A template of an ANOVA table is given in Table 9.14 (cf. Neter *et al.*, 1990), with a = the number of levels of factor A (*Size*: a = 3); b = the number of levels of factor B (*Structure*: b = 2); n = the sample size (the number of subjects for each treatment: n = 94). The sums of squares for each of the dependent variables have been calculated from the data[37].

| Source of Variation | Sum of Squares SS | degrees of freedom df | Mean Squares MS = SS / df |
|---|---|---|---|
| Subjects | SSS | n-1 | MSS = SSS / (n-1) |
| Factor A | SSA | a-1 | MSA = SSA / (a-1) |
| Factor B | SSB | b-1 | MSB = SSB / (b-1) |
| AB Interactions | SSAB | (a-1)(b-1) | MSAB = SSAB / ((a-1)(b-1)) |
| Error | SSE | (n-1)(ab-1) | MSE = SSE /((n-1)(ab-1)) |
| Total | SSTO | abn-1 | |

*Table 9.14 Template ANOVA table for Two-Factor Repeated Measures Design with Repeated Measures on Both Factors*

[35] In SPSS called the studentized deleted residual

[36] Number of outliers on *Time* per treatment: SX 1; SD 2; MX 1; MD 3; LX 0; LD 2.

[37] with SPSS for Windows

In the subsequent sections, the influence of the factors *Size* and *Structure* will be analysed for each of the dependent variables *Time* and *Correctness*:

1. A summary statistic will be given with treatment means and factor level means variable (with sample standard deviations) according to Table 9.7.
2. The treatment means for structured and nonstructured function definitions will be displayed as function of the *Size*.
3. An analysis of variance will be presented according to Table 9.14.
4. The hypotheses on the interaction effects and the factor effects will be tested.

### 9.8.3 Time

For each treatment, the sample mean *Time* in seconds (and the standard deviation) is given in Table 9.15, together with the factor level means. The sample treatment mean $m_{jk} = \Sigma i\ y_{ijk} / n$, where $y_{ijk}$ is the observed value on the dependent variable $Y_{ijk}$.

| Factor | | Size | | | |
|---|---|---|---|---|---|
| | Levels | Small | Medium | Large | Level Mean |
| Structure | Nonstructured X | 37.40 (18.81) | 50.89 (24.95) | 74.96 (35.97) | 54.42 |
| | Structured D | 31.46 (16.97) | 45.80 (18.94) | 64.49 (24.89) | 47.25 |
| | Level Mean | 34.43 | 48.35 | 69.73 | 50.83 |

*Table 9.15 The sample mean Time (seconds) and standard deviation (n=94)*

In Figure 9.4 the sample treatment means of the variable *Time* for structured (D) and nonstructured (X) function definitions are displayed as function of the *Size*.

In case of this continuous dependent variable, we can use the F*-statistic with has the F-distribution under the null hypothesis. The decision rules are as follows:

- Interaction effect AB (Size × Structure) with F* = MSAB / MSE.
  If $F^* \leq F[1-\alpha; (a-1)(b-1), (n-1)(ab-1)]$ we fail to reject the null hypothesis $H_0^1$; otherwise the null hypothesis is rejected and we accept the alternative hypothesis $H_a^1$.
- Main effect factor A (Size) with F* = MSA / MSE.
  If $F^* \leq F[1-\alpha; (a-1), (n-1)(ab-1)]$ we fail to reject the null hypothesis $H_0^2$; otherwise the null hypothesis is rejected and we accept the alternative hypothesis $H_a^2$.

- Main effect factor B (Structure) with F* = MSB / MSE.
  If $F^* \leq F[1-\alpha; (b-1), (n-1)(ab-1)]$ we fail to reject the null hypothesis $H_0^3$; otherwise the null hypothesis is rejected and we accept the alternative hypothesis $H_a^3$.

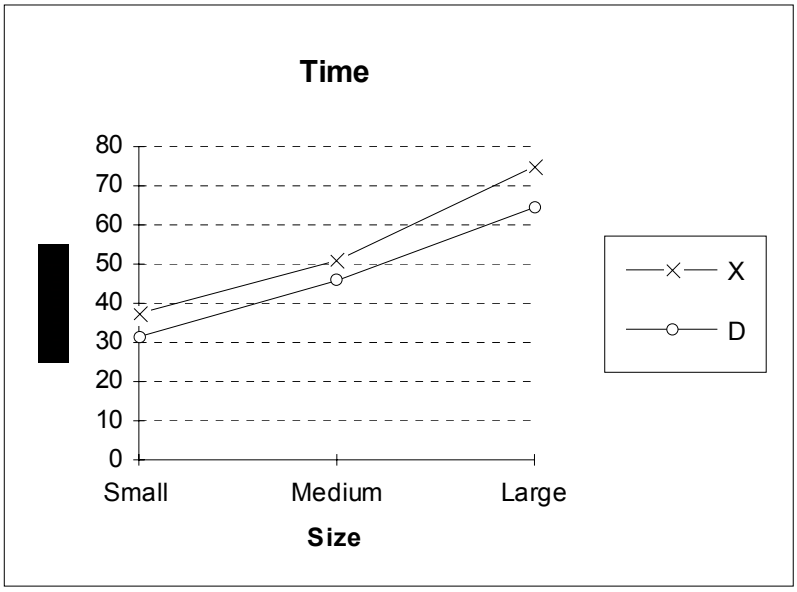



*Figure 9.4 The means of Time (seconds) measured for X- and D-structured scripts as function of the Size*

The ANOVA table for the dependent variable *Time* is given in Table 9.16, together with the calculated F*-value, the F-value at significance level α, and also the p-value (the probability, when $H_0$ is true, of observing a test result as deviant or more deviant than the result actually obtained).

| Source of Variation | Sum of Squares | df | Mean Squares | F* | F α=0.05 | p |
|---|---|---|---|---|---|---|
| Subjects | 96382 | 93 | 1036.37 | | | |
| Factor Size | 118828.1 | 2 | 59414.05 | 117.6 | 3.00 | .000 |
| Factor Structure | 7249.09 | 1 | 7249.09 | 13.68 | 3.84 | .000 |
| Size × Structure Interactions | 783.78 | 2 | 391.89 | 0.820 | 3.00 | .443 |
| Error | 232469.4 | 465 | 499.93 | | | |
| Total | 455712.3 | 563 | | | | |

*Table 9.16 ANOVA table for the variable Time*

From Table 9.16, with a level of significance α = 0.05, it can be concluded that:

- There is no significant interaction effect between *Size* and *Structure* on the variable *Time*, since F* ≤ F (p = .443) (the curves of the treatment means in Figure 9.4 for the two levels of *Structure* are nearly parallel): i.e., we fail to reject the null hypothesis $H_0^1$.
- There is a significant main effect of *Size* on the variable *Time*, since F* > F (p = .000). This means that the null hypothesis $H_0^2$ can be rejected.
- There is a significant main effect of *Structure* on the variable *Time*, since F* > F (p = .000). This means that the null hypothesis $H_0^3$ can be rejected.

The hypotheses $H_0^4$,..,$H_0^{12}$, involving the treatment means, can be tested as well. The family level of significance is chosen to be $\alpha = 0.10$. There are 9 pair-wise comparisons of treatment means, each of them can be analysed with a single degree of freedom test (Neter *et al.*, 1990) with $\alpha' = 0.10/9 = 0.011$. The t* test statistic has been used, with $t^* = (m_{jk}-m_{j'k'}) / \sqrt{(2 \times MSE / n)}$, and degrees of freedom = (n-1)(ab-1). Under the null hypothesis, the statistic t* follows the t distribution. Here, t[0.011;465] = 2.33 .

| Null Hypothesis | Estimated | t* | t* ≥ t | p |
|---|---|---|---|---|
| $H_0^4$ : $\mu_{SX} - \mu_{SD} = 0$ | $m_{SX} - m_{SD} = 5.94$ | 1.82 | False | .036 |
| $H_0^5$ : $\mu_{MX} - \mu_{MD} = 0$ | $m_{MX} - m_{MD} = 5.09$ | 1.56 | False | .070 |
| $H_0^6$ : $\mu_{LX} - \mu_{LD} = 0$ | $m_{LX} - m_{LD} = 10.47$ | 3.21 | True | .001 |
| $H_0^7$ : $\mu_{MX} - \mu_{SX} = 0$ | $m_{MX} - m_{SX} = 13.49$ | 4.14 | True | .000 |
| $H_0^8$ : $\mu_{LX} - \mu_{MX} = 0$ | $m_{LX} - m_{MX} = 24.07$ | 7.38 | True | .000 |
| $H_0^9$ : $\mu_{LX} - \mu_{SX} = 0$ | $m_{LX} - m_{SX} = 37.56$ | 11.52 | True | .000 |
| $H_0^{10}$: $\mu_{MD} - \mu_{SD} = 0$ | $m_{MD} - m_{SD} = 14.34$ | 4.40 | True | .000 |
| $H_0^{11}$: $\mu_{LD} - \mu_{MD} = 0$ | $m_{LD} - m_{MD} = 18.69$ | 5.73 | True | .000 |
| $H_0^{12}$: $\mu_{LD} - \mu_{SD} = 0$ | $m_{LD} - m_{SD} = 33.03$ | 10.13 | True | .000 |

*Table 9.17 Single degree of freedom tests for hypotheses on the variable Time*

From Table 9.17, it can be concluded that in tests 4 and 5 the null hypothesis cannot be rejected; in the other tests (6..12) the null hypothesis can be rejected and the corresponding alternative hypothesis will be accepted. In other words, in these cases there is a significant influence (with a family level of significance $\alpha = 0.10$) on the dependent variable *Time*.

If there had been an interaction effect, the effect of *Size* on the *Structure*-effect could have been tested with the following hypotheses:

$H_0^{13}$: $(\mu_{SX} - \mu_{SD}) - (\mu_{MX} - \mu_{MD}) = 0$

$H_0^{14}$: $(\mu_{MX} - \mu_{MD}) - (\mu_{LX} - \mu_{LD}) = 0$
$H_0^{15}$: $(\mu_{SX} - \mu_{SD}) - (\mu_{LX} - \mu_{LD}) = 0$

These tests could replace the ratio-measure as defined by Scanlan (1989), as will be discussed in section 9.9.

### 9.8.4 Correctness

For each treatment, the sample mean of the variable *Correctness* (and the standard deviation) is given in Table 9.18, together with the factor level means. The sample mean gives the p*roportion* correct answers; this can also be seen as the probability of a correct answer.

| Factor | | Size | | | |
|---|---|---|---|---|---|
| | Levels | Small | Medium | Large | Level Mean |
| Structure | Nonstructured X | 0.660 (0.48) | 0.600 (0.49) | 0.670 (0.47) | 0.643 |
| | Structured D | 0.710 (0.45) | 0.860 (0.35) | 0.940 (0.25) | 0.836 |
| | Level Mean | 0.685 | 0.730 | 0.805 | 0.740 |

*Table 9.18 Proportion correct answers and sample standard deviation (n=94)*

In Figure 9.5 the proportion correct answers for structured (D) and non-structured (X) function definitions are displayed as function of the *Size*.

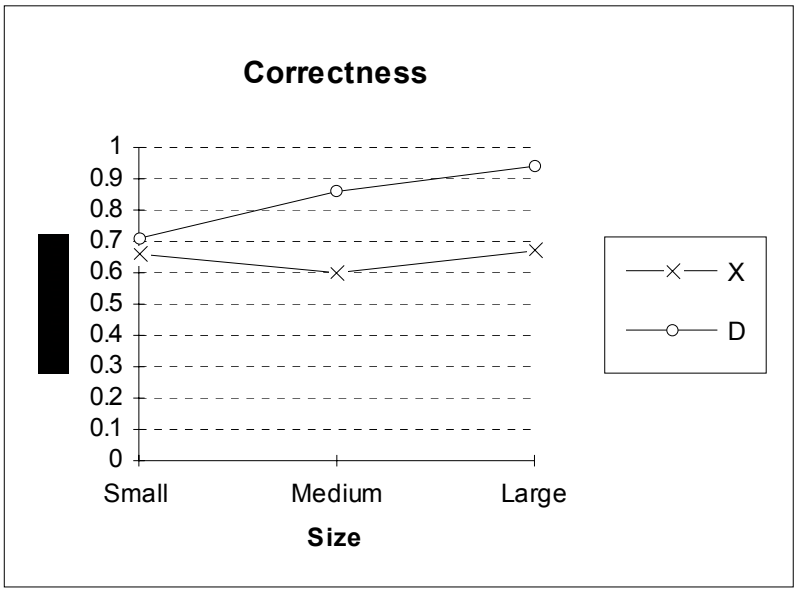



*Figure 9.5 The proportion Correct answers for X- and D-structured scripts as function of the Size*

In case of this binary dependent variable, we will use the Q-statistic, defined by Cochran (Myers *et al.*, 1991), with a $\chi^2$-distribution under the null hypothesis; $\chi^2[1-\alpha, df]$ denotes the $\chi^2$-value at significance level $\alpha$ and df is the degrees of freedom. The decision rules are as follows:

- Interaction effect AB (Size × Structure) with Q = SSAB / MSE.
  If $Q \leq \chi^2[1-\alpha; (a-1)(b-1)]$ we fail to reject the null hypothesis $H_0^1$; otherwise the null hypothesis is rejected and we accept the alternative hypothesis $H_a^1$.
- Main effect factor A (Size) with Q = SSA / MSE.
  If $Q \leq \chi^2[1-\alpha; (a-1)]$ we fail to reject the null hypothesis $H_0^2$; otherwise the null hypothesis is rejected and we accept the alternative hypotheses $H_a^2$.
- Main effect factor B (Structure) with Q = SSB / MSE.
  If $Q \leq \chi^2[1-\alpha; (b-1)]$ we fail to reject the null hypothesis $H_0^3$; otherwise the null hypothesis is rejected and we accept the alternative hypothesis $H_a^3$.

The ANOVA table for the dependent variable *Correctness* is given in Table 9.19, together with the calculated value for the Q-statistic, the $\chi^2$-value at significance level $\alpha$, and the p-value.

| Source of Variation | Sum of Squares | df | Mean Squares | Q | $\chi^2$ $\alpha$=0.05 | p |
|---|---|---|---|---|---|---|
| Subjects | 30.19 | 93 | 0.32 | | | |
| Factor Size | 1.32 | 2 | 0.66 | 7.90 | 5.99 | .021 |
| Factor Structure | 5.36 | 1 | 5.36 | 32.09 | 3.84 | .000 |
| Size × Structure Interactions | 1.42 | 2 | 0.71 | 8.50 | 5.99 | .016 |
| Error | 70.39 | 465 | 0.15 | | | |
| Total | 108.68 | 563 | | | | |

*Table 9.19 ANOVA table for the variable Correctness*

From Table 9.19, with a level of significance $\alpha = 0.05$, it can be concluded that:

- There is a significant interaction effect between *Size* and *Structure* on the variable *Correctness*, since $Q > \chi^2$ (p =.021). This means that the null hypothesis $H_0^1$ can be rejected.
- There is a significant main effect of *Size* on the variable *Correctness*, since $Q > \chi^2$ (p =.000). This means that the null hypothesis $H_0^2$ can be rejected.
- There is a significant main effect of *Structure* on the variable *Correctness*, since $Q > \chi^2$ (p =.016). This means that the null hypothesis $H_0^3$ can be rejected.

Again, we will consider the hypotheses on the treatment means. The family level of significance is chosen to be $\alpha = 0.10$. There are 9 pairwise comparisons of treatment means, each of them can be analysed with a single degree of freedom test with $\alpha' = 0.011$. For each comparison, we can use the McNemar-statistic M with a $\chi^2$-distribution under the null hypothesis. M is estimated as follows (Kotz *et al.,* 1989): $M = (n_{01}-n_{10})^2/(n_{01}+n_{10})$, where $n_{xy}$ is the number of observations having response x on the first treatment in the comparison, and response y on the second treatment (response 0 = incorrect; response 1 = correct). Furthermore, $\chi^2[1-\alpha; df] = \chi^2[0.989; 1] = 6.63$.

| Null Hypothesis | Estimated | M | $M \geq \chi^2$ | p |
|---|---|---|---|---|
| $H_0^4 : \mu_{SX} - \mu_{SD} = 0$ | $m_{SX} - m_{SD} = -0.05$ | 0.86 | False | .353 |
| $H_0^5 : \mu_{MX} - \mu_{MD} = 0$ | $m_{MX} - m_{MD} = -0.26$ | 16.89 | True | .000 |
| $H_0^6 : \mu_{LX} - \mu_{LD} = 0$ | $m_{LX} - m_{LD} = -0.27$ | 17.86 | True | .000 |
| $H_0^7 : \mu_{MX} - \mu_{SX} = 0$ | $m_{MX} - m_{SX} = -0.06$ | 1.06 | False | .304 |
| $H_0^8 : \mu_{LX} - \mu_{MX} = 0$ | $m_{LX} - m_{MX} = 0.07$ | 1.32 | False | .250 |
| $H_0^9 : \mu_{LX} - \mu_{SX} = 0$ | $m_{LX} - m_{SX} = 0.01$ | 0.03 | False | .853 |
| $H_0^{10} : \mu_{MD} - \mu_{SD} = 0$ | $m_{MD} - m_{SD} = 0.15$ | 8.17 | True | .004 |
| $H_0^{11} : \mu_{LD} - \mu_{MD} = 0$ | $m_{LD} - m_{MD} = 0.08$ | 3.27 | False | .071 |
| $H_0^{12} : \mu_{LD} - \mu_{SD} = 0$ | $m_{LD} - m_{SD} = 0.23$ | 16.33 | True | .000 |

*Table 9.20 Single degree of freedom tests for hypotheses on the variable Correctness*

From Table 9.20, it can be concluded that in tests 4, 7, 8, 9 and 11 the null hypothesis cannot be rejected. In the other tests 5, 6, 10 and 12 the null hypothesis can be rejected: there is a significant influence (with a family level of significance $\alpha = 0.10$) on the dependent variable *Correctness*.

## 9.9 Discussion

In the following tables, the results from the previous sections have been summarised. The existence of significant main effects and interaction effects, based on the overall analysis of variance ($\alpha = 0.05$), are given in Table 9.21.

For each of the dependent variables *Time* and *Correctness*, there is an overall significant influence of the factor *Structure* and *Size.* For the variable *Time*, no significant interaction effect has been found; for *Correctness*, a significant interaction effect has been shown.

| Factor<br>Variable | Structure | Size | Structure × Size |
|---|---|---|---|
| Time | Yes | Yes | No |
| Correctness | Yes | Yes | Yes |

*Table 9.21 Existence of significant factor effects and interaction effects*

The results of testing the hypotheses involving treatment means are given in Table 9.22: whether or not the null hypothesis has been rejected (family level of significance $\alpha = 0.10$).

| $H_0$ | $H_0^4$ | $H_0^5$ | $H_0^6$ | $H_0^7$ | $H_0^8$ | $H_0^9$ | $H_0^{10}$ | $H_0^{11}$ | $H_0^{12}$ |
|---|---|---|---|---|---|---|---|---|---|
| | $\mu_{SX}$-<br>$\mu_{SD}$ | $\mu_{MX}$-<br>$\mu_{MD}$ | $\mu_{LX}$-<br>$\mu_{LD}$ | $\mu_{MX}$-<br>$\mu_{SX}$ | $\mu_{LX}$-<br>$\mu_{MX}$ | $\mu_{LX}$-<br>$\mu_{SX}$ | $\mu_{MD}$-<br>$\mu_{SD}$ | $\mu_{LD}$-<br>$\mu_{MD}$ | $\mu_{LD}$-<br>$\mu_{SD}$ |
| Time | No | No | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| Correct | No | Yes | Yes | No | No | No | Yes | No | Yes |

*Table 9.22 Rejection of null hypotheses 4..12 on treatment means*

With these results, it can be seen that:

- The overall significant effect of *Structure* on the variable *Time* appears to be mainly due to the effect of the scripts of size *Large*.
- The overall significant effect of *Structure* on *Correctness* appears to be mainly due to the effect of the Medium and Large scripts.
- The overall significant effect of *Size* on X-structured and D-structured scripts on the variable *Time* is confirmed on each comparison.
- The overall significant effect of *Size* on the variable *Correctness* is mainly due to the effect of *Size* on D-structured scripts. For none of the comparisons on X-structured scripts, a significant influence of the factor *Size* has be shown. This also shows the interaction between the factors *Size* and *Structure* on the variable *Correctness*.

The two dependent variables - *Time* and *Correctness* - have been taken as criteria for the performance of programmers. We assumed that the performance on structured function definitions versus comparable nonstructured function definitions corresponds to the performance on D-structured paths versus the similar X-structured paths as tested in the hypotheses. Then, in summary, we conclude that, with respect to the structure and size of function definitions:

1. subjects need significant *less time* to obtain an answer to structured function definitions than to nonstructured function definitions

2. subjects give significant *more often correct* answers to somewhat larger structured function definitions than to nonstructured function definitions of comparable size
3. subjects need significant *more time* to obtain an answer to larger function definitions than to smaller ones
4. subjects give significant *more often correct* answers for larger structured function definitions than for smaller ones

Conclusions 1 and 2 give empirical evidence to support the general conclusion that programmers perform better on structured Miranda function definitions than on nonstructured definitions.

Conclusion 3 seems to be quit obvious, but conclusion 4 came up as rather a surprise and seems to be counter-intuitive: an increase in the proportion correct answer for larger function definitions. However, a similar 'unexpected' trend has been observed in other studies: Basili & Perricone (1984) found that there is a higher error rate in smaller sized modules than in larger modules. One of the tentative explanations they offer is that larger modules are coded with more care than smaller modules because of their size. Also Möller & Paulish (1993) found significantly higher fault rates in small modules as compared to larger ones.

The experiment used in this study is similar to the one used by Scanlan (1989) in the comparison of flowcharts and pseudocode. However, the design and statistical analysis used in this study differ on some important aspects.

- Scanlan used a single factor repeated measures design, as opposed to a two-factor repeated measures design used here. In our study, the main effects and interaction effect have been established on basis of analysis the variance; the dependency of hypotheses on the treatment means has been accounted for explicitly.
- A ratio-measure is used by Scanlan in order to assess the interaction effect. In terms of the present study, the ratio is calculated by dividing the larger time (of the structured or nonstructured definition) by the smaller time (of the structured or nonstructured definition) for each subject at each size level; those ratios in favour of structured definitions receive a positive sign; those ratios in favour of nonstructured definitions receive a negative sign. In our experiment, the ratio-measure resulted in a highly non-normal distribution, because of the discontinuity of the measure between -1 and +1. Furthermore, in case of equal times, there is no appropriate decision rule to assign the value -1 or +1. In this chapter, an alternative is proposed for the ratio-measure (see section 9.8.3).

- The confidence measure used by Scanlan is on an ordinal scale: four levels from 1 to 4. The mean confidence level of such an ordinal measure, as used by Scanlan, is questionable. Moreover, it is not obvious that the subjects are reliable in the self-assessment of the correctness of their solution, in other words whether the confidence level depends on the correctness of their answer. In some other studies it has been shown that this is not always the case. Gibson & Senn (1989) found a notable discrepancy between correctness and confidence. Gathy and Denef (1993) found a strong correlation between the self-confidence assessment scores and the final examinations for good students, whereas a negative but loose correlation was observed for weak students. Leclercq (1993) analysed factors that affect the confidence estimation and the confidence expression.

With respect to these points, Scanlan's study should be reconsidered.

## 9.10 Conclusion

The aim of this study has been to investigate programmers' performance on structured versus nonstructured function definitions. In the experiment, based on a two-factor repeated measures design, the control-flow model of Miranda function definitions and related metrics proved to be useful in the definition the factors and factor levels of *Structure* and *Size*.

The experimental findings support the main hypothesis that programmers perform better on structured Miranda function definitions than on nonstructured definitions. Some counter-intuitive findings, reported in the literature before, came up in the present study as well: programmers make fewer errors in larger function definitions than in smaller ones.

Based on these experimental findings, the programming style rule can be put forward to use structured function definitions instead of nonstructured ones. This would mean that a programming style is adopted to write guards that always are concluded with an 'otherwise'-case.

```
f (x:xs) = 1, if x > 2
         = 2, otherwise
f [x]    = 3
```

```
g (x:xs) = 1, if x > 2
         = 2, if x ≤ 2
g [x]    = 3
```

*Table 9.23 Example of a structured function definition f and a semantically equivalent nonstructured definition g*

The rule could be relaxed by demanding *total* guards, such that always, once a pattern succeeds, one of the guards in the clause will succeed. In that situa-

tion, if no 'otherwise'-case is used to obtain total guards, a nonstructured function definition would be obtained in the control-flow model, because in the model is abstracted from the actual content of the guards. An example is given in Table 9.23. Both definitions have total guards. In the control-flow model, the function definition of *f* is structured, whereas the semantically equivalent definition of *g* is nonstructured.

To check the application of this programming rule, a Miranda static analyser (van den Berg & van den Broek, 1995a) based on the control-flow model of function definitions can be used. With this analyser, X-structured function definitions in scripts can be spotted easily, also in scripts with many definitions. After this anomaly checking, these definitions can be inspected on errors and/or be rewritten to a structured version.

In a survey of scripts written by experts, hardly any nonstructured function definition has been found. Apparently, experts already do not use this kind of function definitions. For some programmers with a few years of functional programming experience, nonstructured function definitions have been detected in their scripts. Programming style rules, as proposed above, could make programmers aware of the operational semantics of function definitions.

## Acknowledgement

The authors would like to thank N. Fenton for his comments on an earlier version of this chapter; R. Houterman and S. Oosterloo for their advise on the statistics used in this chapter; E. Prangsma and M. Harssema for the assistance in the experiment; and last but not least, the students for their participation in the experiment.

# Conclusion

In the general introduction of the thesis, the following problems have been put forward:

- How can aspects of software quality in different programming paradigms be assessed using software metrics.
- How can software in a functional programming language be modelled to capture structural properties of this software.
- How can software models and metrics be validated in experiments based on measurement theory.

As the end result of the thesis, one would expect a conclusion such as:

> *The results of the experiments show that the programs in a functional language took significantly less time to develop and were considerably more concise and easier to understand than the corresponding programs written in an imperative language.*

However, it is hard to reach such a general conclusion on objective grounds. In this thesis, based on the experiment described in Chapter 2, a tentative conclusion is put forward, that students in the experimental group who learned functional programming (FP) made 'better' programs than students in the control group who learned imperative programming (IP). This type of conclusion raised the question of its validity (as discussed in part C of this thesis).

The conclusion in Chapter 2 has been based on one particular type of assignments: the design and implementation of a program. An important criterion used in the experiment is the coverage of the design by its implementation. It appeared that students in the FP-group showed a higher coverage than students in the IP-group. This could be rephrased as follows: FP-students used more abstractions in their programs than IP-students. The type of abstractions are functional (or procedural) abstractions. Hence, the conclusion is as follows: FP-students used more functions in their programs than IP-students. However, this is hardly surprising, since the FP-students used a *functional* programming language in which functions are more prominent than in imperative

languages. This confounding effect may obscure the actual differences between the two programming groups.

In Chapter 3, another problem arose: experts were requested to rank students programs on some given criteria. Apparently, for functional programs there was less consensus than for imperative programs, because of the novelty of the functional programming style in teaching. This points to another major problem in this type of research: the importance of differences between programmers:

> *The variability due to subject differences often outweighs variability due to independent variables. (Moher & Schneider, 1982)*

This problem can be tackled only by careful experimentation. This thesis has been aimed specifically at the issue of experimentation: in the second part, the modelling of software for structure metrics has been described; and in part C, the validation of these metrics has been investigated in some experiments.

One important lesson learned in the previous years in software measurement is that 'no single technology or method can be expected to work well in all contexts, and observing software phenomena out of context seems to be doomed to fail.' (Basili *et al.,* 1993).

Therefore, there will no general answer to the question in Chapter 1: 'is functional programming the best initial programming language', but the question should be: 'is functional programming the best initial programming language, for a given group of students, with a certain educational background, with a certain motivation, within a certain curriculum, and so on.' This situation can be made more concrete: the population of first-year students in the department has changed over the last few years: now, not only the 'pure' Computer Science students take the functional programming course, but also students in Business Information Technology. These students differ in several aspects from the original group. Moreover, in several courses in the curriculum, there is a tendency to emphasise an object orientation to programming and design. Again, this may have an impact on the answer to the question regarding the best initial programming language.

This leads to another conclusion: 'We need to characterise and understand the project context and understand the various phenomena relative to that context ... . We need to replicate experiments in different contexts to fully understand the nature of the various phenomena ... . ' (Basili *et al.,* 1993).

The context for the research in this thesis can be characterised by programming-in-the-small, which has been investigated in controlled experiments with novice programmers.

Although there are no general conclusions such as stated in the previous section, there are other achievements of the research presented in this thesis.

It has been shown (Chapters 4 and 5) that some models, e.g. flowgraphs and callgraphs, can be used to model software in different programming paradigms. These models, which are mainly used for imperative programming, have been adapted for programs in the functional programming language Miranda. The proposed control-flow model captures the operational semantics of function definitions. The callgraph model has been adapted to specific properties of functional programs. The models used for both programming paradigms allow comparison of software in different languages based on these abstractions and related metrics. However, the validation of these comparisons is a major problem to be solved, as has been shown for the McCabe and Halstead metrics in some exploratory experiments with respect to the comprehensibility (Chapters 3 and 2).

The objectivity of the assessment of software attributes has been enhanced by the use of static analysis tools, by which the measurement is automated. For this purpose, the metrics based on these models have been formalised in this research by means of attributed grammars (Chapters 3 and 5). Somewhat larger software has also been analysed with the metrics tools. These analysers are a helpful supplement to other approaches of static analysis.

The validation of structure metrics has been addressed in a study of Miranda type expressions (Chapter 6). Structure metrics have been defined on the base of parse trees of type expressions. A framework for validation emerged in this case study. In order to substantiate the validation of software metrics, the role of measurement theory has been investigated (Chapter 7). Representation axioms have been used in establishing the empirical and theoretical order of type expressions. For some subsets of type expressions in the experiment, there is a good correspondence between the empirical order and the hypothesised theoretical order. Different types of validities in the software metric development process have been clarified by combining a validation network scheme and the representational measurement theory (Chapter 8).

In an extended experiment, the control-flow model for Miranda function definitions has been utilised to establish the influence of the structure of definitions on their comprehensibility (Chapter 9). The experimental design and the statistical analysis have been described in detail. Structured function definitions appear to be easier to understand by novice programmers than non-structured ones. A programming style rule on the use of guards in Miranda function definitions has been validated by these findings.

## Further research

The research presented in this thesis could be extended in many directions. Few of them will be brought to the fore: firstly, in the field of experimental software engineering, and secondly, those with some implications for the computer science curriculum.

There is a need for metrics of products in the early phases of software development, e.g. specification and design, and for a relation between these metrics and the characteristics of the final software products. Functional programs are being used as executable specifications. The models and tools described in this thesis could be investigated in this context.

A database ought to be set up of metrics of functional programs, with data on development effort, error rates, and such-like, to highlight characteristics of functional programs in different development environments.

The actual application of metric tools in software development using functional programming, particularly with respect to testing and maintenance, should be investigated in order to tailor the metrics and the tools to an optimal use in specific environments.

Controlled experiments, as described in this thesis, should be extended to validate other rules in functional programming, and the role of measurement theory in these experiments should be strengthened.

In computer science education, the use of software metrics for the objective assessment of programming assignments could be pursued (cf. Ceilidh: Benford *et al.,* 1994). Probably as important is that students in computer science get acquainted with software measurement to support the development and management of software. This could be achieved by the use of metric tools in programming courses and programming projects. Furthermore, the curriculum could be extended with courses specifically directed to software measurement and experimentation:

> *Measurement and experimentation are standard ingredients in traditional science and engineering curricula. ... Recurring use of this paradigm within the [computer science] curriculum is important to facilitate its being learned properly. (Zweben, 1993)*

Some extended packages for software measurement education have been developed (e.g. Metkit: Ashley, 1994).

The thesis will be concluded with a quotation of Rombach, stated in the preface to a Workshop on Experimental Software Engineering:

> *We have only begun to understand the experimental nature of software engineering, the role of empirical studies and measurement within software engineering, and the mechanisms needed to apply them successfully. (Rombach et al., 1993)*

The research in this thesis will have achieved its goal when it has made a contribution to this understanding.

# Summary

In general, a producer is interested in the quality of his product, whether it is a software package or, for example, a car. There are quality aspects which are important to the user of the product, such as for a car the fuel consumption rate. Other quality aspects are relevant to the technicians who have to build the product or to maintain it: e.g. the ease of assembling certain parts. Furthermore, the producer will be interested in the cost and duration of the production, and the resources needed. Such quality aspects have to be measured to allow a comparison with other products and production processes: a particular fuel consumption rate will be acceptable in certain circumstances.

A similar situation is encountered in the case of software. There are user aspects of quality, for example with respect to the interface and performance, and other aspects related to the programmers who have to design and implement the computer programs. The discipline of software engineering offers methods for the design and production of software. The field of software measurement provides approaches to the quantification of quality aspects of software, related to the product, the process and the resources. An obvious software metric is the size of the program, usually expressed in the number of lines of executable code. But there are many other software metrics, and it is necessary to be able to decide when to use which metric and how. With these metrics, one would like to be able to make an objective assessment of the relative merits of software products and software development methods.

This thesis addresses some issues on the quality of software with respect to the programmers: the comprehensibility of the program code. A lot of time is spent reading and understanding programs in order to remove faults or to adapt the program to changed requirements. Many factors in the program code affect the comprehensibility of the program, such as the language used, the naming of variables, the structure, the indentation, explanatory documentation, the experience of the programmer, and so on.

In order to capture a particular quality aspect of programs, usually a model is built. In such models, certain details in the program are abstracted. The

models are used in the definition of software metrics. The models and metrics have to be validated, for example their consistency has to be established. Furthermore, the metric values can be obtained with the use of tools: i.e. another computer program is used to analyse the original programs. The tools assure a fixed procedure and thus an objective assessment of the quality aspects.

This thesis focuses on the structure of the code, how it is divided into parts - usually called modules - and how the modules are related to each other. This aspect of structure is modelled in a callgraph of the program. Another aspect studied in this thesis is the control structure: the order in which parts of the program will be executed, as prescribed by special language constructs. For this aspect, a control-flow graph of the program is used. The metrics are indicators of the complexity of the structure. These models and metrics are described in Chapters 4 and 5. A tool for the automated measurement of metrics based on these models is described in Chapter 5.

Two classes of programming languages are considered: the 'classical' imperative ones, with languages such as Pascal and Modula-2, and the less common class of functional languages, where Miranda is used as the example. The latter is a very powerful mathematics-like language. These languages are used in the initial programming courses in Computer Science at the University of Twente as described in Chapter 2. One would like to know whether students who learn to program in Miranda write better programs than the students who learn for example Modula-2; and also: are these Miranda programs easier to comprehend than Modula programs? For this comparison, some experiments with certain well-known software metrics are described in Chapters 2 and 3. Some models, the callgraph and the control-flow graph, that are used for imperative languages, are modified for the functional language Miranda as described in Chapter 5.

Once one has obtained metric values, it has to established how they can be used. Do they yield the expected ordering of programs, e.g. with respect to their comprehensibility? Are there threshold values beyond which the programs are difficult to understand, or are very error prone? These questions are part of the external validation. The validation has been carried out in some formal experiments using small programs with first-year students, thus novice programmers. They are described in Chapters 6 and 9. The use of measurement theory in the validation is explored in Chapter 7. It is an open question whether the results of experiments involving novice programmers and small programs can be generalised to expert programmers in the industry working on large programs in teams. Several of these validation issues are raised in Chapter 8 of this thesis.

# References


Akker, R. op den (Ed.), Hoeven, G. van der, Joosten, S. & Seters, H. van (1992). *Functioneel Programmeren, studentenhandleiding* [Functional Programming, student's manual]. Enschede: University of Twente.

Akker, R. op den (Ed.), Hoeven, G. van der, Joosten, S. & Seters, H. van (1992b). *Functioneel Programmeren, docentenmateriaal* [Functional Programming, teacher's manual]. Enschede: University of Twente.

American Psychological Association (1954). *Technical Recommendations for Psychological Tests and Diagnostic Techniques*. Washington: American Psychological Association.

Ashley, N. (1994). METKIT: Training in How to Use Measurement as a Software Management Tool. *Software Quality Journal, 3,* 129-136.

Bache, R. & Bazzana, G. (1994). *Software Metrics for Product Assessment*. London: McGraw-Hill.

Bache, R. & Leelasena, L. (1990). *Qualms, A Tool for Control Flow Analysis and Measurement*. CSSE, London: South Bank Polytechnic.

Bache, R. & Wilson, L. (1988). Details of the Implementation of the Decomposition Algorithm. In: J.J. Elliott, N.E. Fenton, S. Linkman, G. Markman, & R. Whitty (Eds): *Structure-based Software Measurement*. London: CSSE, South Bank Polytechnic, 270-284.

Bache, R.M. (1990). *Graph Models of Software*. PhD Dissertation, London: Southbank Polytechnic.

Backus, J. (1978). Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications ACM 21(8),* 613-641.

Bailes, P.A. & Salzman, E.J. (1989). A Proposal for a Bachelors Degree Program in Software Engineering. *Software Engineering Education.* Lecture Notes in Computer Science 376, 90-108. Berlin: Springer.

Bailey, R. (1990). *Functional Programming with HOPE.* Chichester: Ellis Horwood.

Baker, A.L., Bieman, J.M., Fenton, N., Gustafson, D.A., Melton, A. & Whitty, R. (1990). A Philosophy for Software Measurement. *J. Systems Software, 12,* 277-281.

Baker, E. L., Atwood, N. K. & Duffy, T. M. (1988). Cognitive Approaches to Assessing Readability of Text. In: A. Davison & G.M. Green (Eds): *Linguistic Complexity and Text Comprehension: Readability Issues Reconsidered, 55-84.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Basili, V.R. & Perricone, B.T. (1984). Software Errors and Complexity: an Empirical Investigation. *Communications ACM, 27(1),* 42-52.

Basili, V.R., & Rombach, H.D. (1988). The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Trans. Softw. Eng., SE 14*, 758-773.

Basili, V.R., Selby, R.W. & Hutchens, D.H. (1986). Experimentation in Software Engineering. *IEEE Trans. Softw. Eng., SE 12 (7),* 733-743.

Benford, S., Burke, E. & Foxley, E. (1994). *Courseware to Support the Teaching of Programming*. Report Dept. Computer Science, University of Nottingham.

Berg, K.G. van den & Broek, P.M. van den (1994). Axiomatic Testing of Structure Metrics. *Proceedings of the Second International Software Metrics Symposium*. London: IEEE Computer Society Press, 45-53. (This thesis Chapter 7).

Berg, K.G. van den & Broek, P.M. van den (1995a). Static Analysis of Functional Programs. *Information and Software Technology, 37(4),* 213-224. (This thesis Chapter 5).

Berg, K.G. van den & Broek, P.M. van den (1995b). Programmers' Performance on Structured versus Nonstructured Function Definitions. *Memoranda Informatica 95-11*, Enschede: University of Twente. (This thesis Chapter 9).

Berg, K.G. van den & Broek, P.M. van den (1995c). Axiomatic Validation in the Software Metric Development Process. In: A. Melton (Ed.): *Software Measurement: Understanding Software Engineering*, London: Thomson. (This thesis Chapter 8).

Berg, K.G. van den & Pilot, A. (1989). Functioneel Programmeren 1987/88. Evaluatie van een onderwijsexperiment [Functional Programming 1987/88. Evaluation of an educational experiment], *Memoranda Informatica INF-89-6.* Enschede: University of Twente.

Berg, K.G. van den (1992). Syntactic Complexity Metrics and the Readability of Programs in a Functional Computer Language. In: F.L. Engel, et al. (Eds): *Cognitive Modelling and Interactive Environments in Language Learning*, NATO Advanced Science Institute Series. Berlin: Springer, 199-206. (This thesis Chapter 3).

Berg, K.G. van den (1992a). *Imperatief Programmeren,* [Imperative Programming] Lecture notes. Enschede: University of Twente.

Berg, K.G. van den, Broek, P.M. van den & Petersen, G.M. van (1993). Validation of Structure Metrics: A Case Study. *Proceedings of International Software Metrics Symposium METRICS 93.* Washington: IEEE Computer Society Press, 92-99. (This thesis Chapter 6).

Berg, K.G. van den, Massink, M. & Pilot, A. (1989). *Experimentele Vergelijking van het Leren Programmeren ondersteund door een Functionele versus een Imperatieve Programmeertaal.* [Experimental comparison of programming education supported by a functional versus an imperative programming language]. Leiden: Educational Research Days.

Berka, K. (1983). *Measurement: Its Concepts, Theories, and Problems*. Dordrecht: Reidel.

Berne, H. van, Duijvestijn, A.J.W. & Hoeven, G.F. van der (1985). Functionele Talen [Functional Languages]. *Informatie 27(10),* 837-928.

Bieman, J., Fenton, N.E., Gustafson, D., Melton, A. & Whitty, R. (1992). Moving from Philosophy to Practice in Software Measurement. In: T. Denvir, R. Herman & R.W. Whitty (Eds): *Formal Aspects of Measurement*. London: Springer, 38-59.

Bird, R.S. & Wadler, Ph. (1988). *Introduction to Functional Programming*. New York: Prentice Hall.

Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J. & Merritt, M.J. (1973). *Characteristics of Software Quality*. Software Series 73-09. Redondo Beach (TRW). Also published in 1978, Amsterdam: North Holland.

Brinberg, D. & McGrath J.E. (1985). *Validity and the Research Process*. Newbury Park: Sage.

Broek, P.M. van den & Berg, K.G. van den (1993). Modelling Software for Structure Metrics, *Memoranda Informatica 93-12.* Enschede: University of Twente. (This thesis Chapter 4).

Broek, P.M. van den & Berg, K.G. van den (1995). Generalised Approach to Structure Metrics. *Software Engineering Journal, 10(2),* 61-67.

Bush, M.E. & Fenton, N.E. (1990). Software Measurement: A Conceptual Framework. *J. Systems Software, 12,* 223-231.

Cantor, G. (1895). Beiträge zur Begründung der transfiniten Mengenlehre. *Mathematische Annalen, 46*, 481-512.

Card, D. N. & Glass, G. L. (1990). *Measuring Software Design Quality*. Englewood Cliffs, NJ: Prentice Hall.

Cardelli, L. & Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys, 17 (4),* 471-522.

Clack, C., Myers, C., & Poon, E. (1995). *Programming with Miranda*. New York: Prentice Hall.

Conte, S. D., Dunsmore, H. E. & Shen, V. Y. (1986). *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings.

Coulter, N. S. (1983). Software Science and Cognitive Psychology. *IEEE Trans. Softw. Eng., SE 9(2),* 166-171.

Curtis, B. (1981). The Measurement of Software Quality and Complexity. In: A. Perlis, F. Sayward & M. Shaw (Eds): *Software Metrics: An Analysis and Evaluation, 203-223.* Cambridge, Mass: MIT Press.

Curtis, B. (1986). *Human Factors in Software Development*. 2nd ed. Washington: IEEE Computer Society.

Curtis, B., Sheppard, S. B., Milliman, P., M. A. & Love, T. (1979). Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Trans. Softw. Eng., SE 5(2),* 96-104.

Davies, S. P. (1989). Skill Levels and Strategic Differences in Plan Comprehension and Implementation in Programming. In: A. Sutcliffe & L. Macaulay (Eds): *Peoples and Computers V (487-502).* Cambridge: Cambridge University Press.

Davies, S.P. (1993). Models and Theories of Programming Strategy. *Int. J. Man-Machine Studies,* 39, 237-267.

Fenton, N. E. & Kaposi, A. A. (1989). An Engineering Theory of Structure and Measurement. In: B. A. Kitchenham & B. Littlewood (Eds): *Measurement for Software Control and Assurance.* London: Elsevier, 335-384.

Fenton, N.E. & Kaposi, A.A. (1987). Metrics and Software Structure*, Information and Software Technology, 29(6),* 301-320.

Fenton, N.E. (1991). *Software Metrics: A Rigorous Approach*. London: Chapman & Hall.

Fenton, N.E. (1992). When a Software Measure is not a Measure. *Software Engineering Journal, Sept, 357-362.*

Fenton, N.E. (1994). Software Measurement: A Necessary Scientific Basis. *IEEE Trans. Softw. Eng. SE 20 (3),* 199-206.

Fenton, N.E., Pfleeger, S.L. & Glass, R.L. (1994). Science and Substance: A Challenge to Software Engineers. *IEEE Software*, July, 86-95.

Finkelstein, L., & Leaning, M.S. (1984). A Review of the Fundamental Concepts of Measurement. *Measurement*, 2(1), 25-34.

Fleck, A. C. (1990). A Case Study Comparison of Four Declarative Programming Languages. *Software-Practice and Experience, 20 (1)*, 49-65.

Fodor, J.A., Bever, T.G., & Garret, M.F. (1974). *The Psychology of Language*. New York: McGraw-Hill.

Frazier, L. (1988). The Study of Linguistic Complexity. In: A. Davison & G.M. Green (Eds): *Linguistic Complexity and Text Comprehension: Readability Issues Reconsidered, 193-221*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Gathy, P. & Denef, J.-F. (1993). Self-Assessment During Computer-Assisted Testing in Histology. In: D.A. Leclercq & J.E. Bruno (Eds): *Item Banking: Interactive Testing and Self-Assessment*. Berlin: Springer, 233-241.

Gibson, V.R. & Senn, J.A. (1989). System Structure and Software Maintenance Performance. *Communications ACM, 32(3),* 347-358.

GrammaTech (1993). *The Synthesizer Generator Reference Manual*, Release 4.1. New York: GrammaTech.

Green, T. R. G. & Borning, A. (1990). The Generalised Unification Parser: Modelling the Parsing of Notations. In: D. Diaper et al. (Eds): *Human-Computer Interaction-INTERACT '90*, 951-957. Amsterdam: North-Holland.

Green, T.R.G. (1980). IF's and THEN's: Is Nesting just for the Birds? *Software-Practice and Experience, 10*, 373-381.

Guilford, J.P., Fruchter, B. (1978). *Fundamental Statistics in Psychology and Education*, London: McGraw-Hill.

Gustafson, D.A., Toledo, R.M., Courtney, R.E. & Temsamani, N. (1992). A Critique of Validation/Verification Techniques for Software Development Measures. In: T. Denvir, R. Herman & R.W. Whitty (Eds): *Formal Aspects of Measurement*. London: Springer, 145-156.

Guttman, L. (1971). Measurement as Structural Theory. *Psychometrika, 36*, 329-347.

Halstead, M. H. (1977). *Elements of Software Science*. New York: Elsevier.

Harrison, R. (1993a). A Declarative Development Technique. *Software Quality Management Conference, Southampton,* 431-444.

Harrison, R. (1993b). Quantifying Internal Attributes of Functional Programs. *Information and Software Technology*, *35(10),* 554-560.

Henderson, P. (1986). Functional Programming, Formal Specification, and Rapid Prototyping. *IEEE Softw. Eng., SE 12(2),* 241-250.

Henry, S. & Goff, R. (1989). Complexity Measurement of a Graphical Programming Language. *Software-Practice and Experience, 19(11)*, 1065-1088.

Holyer, I. (1991). *Functional Programming with Miranda*. London: Pittman.

Hudak, P. & Fasel, J.H. (1992). A Gentle Introduction to Haskell. *ACM Sigplan Notices, 27(5),* T1-T53.

Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, *21(3),* 359-411.

Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal, 32(2),* 98-107.

Ince, D. (1989). *Software Metrics*. In: Kitchenham, B.A. & Littlewood, B. (Eds): Measurement for Software Control and Assurance. London: Elsevier.

Jensen Partners International (1988). *TopSpeed(TM) Modula-2*.

Joosten, S.M.M. & Berg, K.G. van den (1990). Can Computer Programming be based on Functional Programming. *Memoranda Informatica 90-46*. Enschede: University of Twente. (a previous version of Chapter 2 in this thesis).

Joosten, S.M.M. (1989). *The use of Functional Programming in Software Development*. Dissertation, Enschede: University of Twente.

Joosten, S.M.M. (Ed.), Berg, K.G. van den & Hoeven, G.F. van der (1993). Teaching Functional Programming to First-Year Students. *Journal of Functional Programming, 3(1),* 49-65. (This thesis Chapter 2).

Kaposi, A.A. (1990). Measurement Theory. In: J.A. McDermid (Ed.): *Software Engineer's Reference Book*. Oxford: Butterworth/ Heinemann, Ch 12.

Khalil, O. E. & Clark, J. D. (1989). The Influence of Programmer's Cognitive Complexity on Program Comprehension and Modification. *Int. Journal Man-Machine Studies, 31,* 219-236.

Kitchenham, B.A., Linkman, S.G. & Law, D.T. (1994). Critical Review of Quantitative Assessment. *Software Engineering Journal*, March 1994, 43-53.

Koffman, E.B. (1988). *Problem Solving and Structured Programming in Modula-2*. Reading, Massachusetts: Addison-Wesley.

Kosky, A. (1988). Declarative Languages for Advanced Information Technology. *Journal Inf. Techn., 3(2)*, 110-118.

Kotz, S. & Johnson, N.L. (Eds) (1989). *Encyclopaedia of Statistical Sciences*. New York: Wiley.

Krantz, D.H., Luce, R.D., Suppes, P., & Tversky, A. (1971). *Foundations of Measurement*, Volume I. New York: Academic Press.

Leclercq, D.A. (1993). Validity, Reliability, and Acuity of Self-Assessment in Educational Testing. In: D.A. Leclercq & J.E. Bruno (Eds): *Item Banking: Interactive Testing and Self-Assessment*. Berlin: Springer, 114-131.

Lindeman, R.H., Merenda, P. F. & Gold, R. Z. (1980). *Introduction to Bivariate and Multivariate Analysis*. Glenview, Ill.: Scott, Foresman and Company.

Luce, R.D., Krantz, D.H., Suppes, P., & Tversky, A. (1990). *Foundations of Measurement*. Volume III. San Diego: Academic Press.

Maki, D.P & Thompson M. (1973). *Mathematical Models and Applications*. Englewood Cliffs: Prentice-Hall.

Mayrhauser, A. von (1994). Maintenance and Evolution of Software Products, In: M.C.Yovits (Ed.): *Advances in Computers, 39*, 1-49.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Trans. Softw. Eng., SE 2(4),* 308-320.

McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications ACM*, *3(4),* 184-195.

McNamara, T.P., Miller, D.L. & Bransford, J.D. (1991). Mental Models and Reading Comprehension. In: R. Barr, M.L. Kamil, P.B. Mosenthal & P.D. Pearson (Eds): *Handbook of Reading Research*, Vol. II. New York: Longman.

Melton, A. (1992). Specifying Internal, External, and Predictive Software Metrics, In: T. Denvir, R. Herman & R.W. Whitty (Eds): *Formal Aspects of Measurement*, London: Springer, 194-208.

Melton, A.C., Gustafson, D.A., Bieman, J.M. & Baker, A.L. (1990). A Mathematical Perspective for Software Measures Research. *Softw. Eng. Journal*, Sept, 246-254.

Michaelson, G. (1989). *An Introduction to Functional Programming through Lambda Calculus*. Wokingham: Addison-Wesley.

Moerkerke, G., Roossink, H.J., Diepen, N.M. van, Berg, K.G. van den, Dijk, E.M.A.G. van & Koppelman, H., (1992). Over Criteria voor het Beoordelen van Tentamenuitwerkingen in het vak Programmeren, [On criteria for the assessment of programming assignments]. In: F.Mulder (Ed.): *Congresbundel NIOC '90*. Deventer: Kluwer, 548-558.

Moher, T. & Schneider, G.M. (1982). Methodology and Experimental Research in Software Engineering. *Int. J. Man-Machine Studies, 16,* 65-87.

Möller, K-H. & Paulish, D.J. (1993). An Empirical Investigation of Software Fault Distribution, *1st International Software Metrics Symposium*. Baltimore, Washington: IEEE, 82-90.

Myers, C., Clack, C. & Poon, E. (1993). *Programming with Standard ML*. New York: Prentice Hall.

Myers, J.L. & Well, A.D. (1991). *Research Design and Statistical Analysis*. New York: HarperCollins.

Neter, J., Wasserman,W. & Kutner, M.H. (1990). *Applied Linear Statistical Models. Regression, Analysis of Variance, and Experimental Designs*. 3rd ed. Homewood: Irwin.

Parnas, D.L. (1972). On the Criteria to be used in Decomposing Systems into Modules. *Communications ACM, 14(1),* 1053-1058.

Petersen, G.M. van (1992). *Validation of axiomatic structure metrics for comprehensibility of Miranda type expressions*. M.Sc. Thesis, Enschede: University of Twente.

Petre, M. & Winder, R. (1990). On Languages, Models and Programming Styles. *The Computer Journal, 33(2),* 173-180.

Peyton Jones, S.L. (1987). *The Implementation of Functional Programming Languages*. New Jersey: Prentice-Hall.

Plasmeijer, R. & Eekelen, M. van (1993*). Functional Programming and Parallel Graph Rewriting*. Wokingham: Addison-Wesley.

Pomberger, G. (1984). *Software Engineering and Modula-2*. Englewoods Cliffs, Prentice-Hall.

Pressman, R.S. (1992). *Software Engineering, A Practitioner's Approach*. 3rd ed. New York: McGraw-Hill.

Prometrix (1993). *User and Installation Manual*. Glasgow: Infometrix Software.

Qualms (1988). Wilson, L. & Leelasena, L., *The Qualms Program Documentation*, Alvey Project SE/69. London: South Bank Polytechnic.

Reps, T. W. & Teitelbaum, T. (1989). *The Synthesizer Generator: a System for Constructing Language-based Editors*. New York: Springer.

Robbers, R.M.R. (1990). Software Metric Analysers based on Attribute Grammars. The Metrics of Halstead and McCabe for Pascal and Miranda programs. *Memoranda Informatica 90-88*. Enschede: University of Twente.

Roberts, F.S. (1979). *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences*. Encyclopaedia of Mathematics and Its Applications, Volume 7. London: Addison-Wesley.

Robson, D.J., Bennett, K.H., Cornelius, B.J. & Munro, M. (1991). Approaches to Program Comprehension. *J. Systems Software, 14,* 79-84.

Rombach, H.D., Basili, V.R. & Selby, R.W. (Eds) (1993). *Experimental Software Engineering: Critical Assessment and Future Directions*. Berlin: Springer Verlag.

Sammet, J.E. (1981). High Level Language Metrics. In Perlis, A., Sayward, F. & Shaw, M. (Eds): *Software Metrics: An Analysis and Evaluation, 131-141.* Cambridge, Mass: MIT Press.

Samson, W. B., Dugard, P. I., Nevill, D. G., Oldfield, P. E. & Smith, A. W. (1989). The Relationship between Specification and Implementation Metrics. In: B. A. Kitchenham & B. Littlewood (Eds): *Measurement for Software Control and Assurance, 335-384.* London: Elsevier.

Sanders, P. (1989). An Evaluation of Functional Programming for the Commercial Environment. *Br. Telecom. Technol. Journal, 7(3),* 25-33.

Savitch, W.J. (1989). Functional Programming in Pascal? *Journal of Pascal, Ada Modula-2*, *8(5),* 35-41.

Scanlan, D.A. (1989). Structured Flowcharts outperform Pseudocode: an experimental comparison. *IEEE Software, Sept.,* 28-36.

Schach, S.R. (1990). *Software Engineering*. Boston: Irwin & Aksen.

Schneidewind, N.F. (1992). Methodology for Validating Software Metrics. *IEEE Trans. Softw. Eng., SE 18(5),* 410-422.

Schwager, W. (1988). *Theories of Measurement in Social Science: A Critical Review.* Thesis, Rotterdam: Erasmus University.

Shen, V. Y., Conte, S. D. & Dunsmore, H. E. (1983). Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support. *IEEE Trans. Softw. Eng., SE 9(2),* 155-165.

Shepperd, M. & Ince, D. (1991). *Algebraic Validation of Software Metrics*, Lecture Notes in Computer Science 550. Berlin: Springer, 343-363.

Shepperd, M. & Ince, D. (1993). *Derivation and Validation of Software Metrics*. Oxford: Clarendon.

Shepperd, M. & Ince, D.C. (1994). A Critique of Three Metrics. *J. Systems Software, 26,* 197-210.

Springer, G. & Friedman, D.P. (1990). *Scheme and the Art of Programming.* Cambridge, MA: MIT / New York: McGraw-Hill.

Suppes, P., Krantz, D.H., Luce, R.D., & Tversky, A. (1989). *Foundations of Measurement*, Volume II. New York: Academic Press.

Tian, J. & Zelkowitz, M.V. (1992). A Formal Program Complexity Model and its Application. *J. Systems Software, 17,* 253-266.

Turner, D. (1982). Recursion equations as a programming language. In: Darlington, J (Ed.): *Functional Programming and its Applications*. Cambridge: Cambridge University Press, 1-28.

Turner, D.A. (1979). A New Implementation Technique for Applicative Languages. *Software - Practice and Experience, 9,* 31-49.

Turner, D.A. (1985). Functional Programs as Executable Specifications. In: C.A.R. Hoare & J.C. Shepherdson (Eds): *Mathematical logic and programming languages*. Englewood Cliffs: Prentice Hall, 29-54.

Turner, D.A. (1986). An Overview of Miranda. *Sigplan Notices, 21(12),* 158-166.

Ullman, J.D. (1994). *Elements of ML Programming*. Englewood Cliffs: Prentice Hall.

Vessey, I. & Weber, R. (1984). Research on Structured Programming: An Empiricist's Evaluation. *Trans. Softw. Eng., SE 10(4),* 397-407.

Watt, D.A. (1990). *Programming Language Concepts and Paradigms*. New York: Prentice Hall.

Weyuker, E. J. (1988). Evaluating Software Complexity Measures. *IEEE Trans. Softw. Eng., SE 14(9),* 1357-1365.

Whitty, R. (1988). Modelling Sequential Processes for Complexity Measurement, In: J.J. Elliott, N.E. Fenton, S. Linkman, G. Markman & R. Whitty (Eds): *Structure-based Software Measurement*. London: CSSE, South Bank Polytechnic, 185-196.

Whitty, R.W. (1992). Multi-dimensional Software Metrics, In: T. Denvir, R. Herman & R.W. Whitty (Eds): *Formal Aspects of Measurement*. London: Springer, 116-141.

Wikstrom, Å. (1987). *Functional Programming using Standard ML*. Hemel Hempstead: Prentice Hall.

Yourdon, E. & Constantine, L.L. (1979). *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs: Prentice-Hall.

Zuse, H. (1991). *Software Complexity: Measures and Methods*. Berlin: De Gruyter.

Zuse, H. (1992). Properties of Software Measures. *Software Quality Journal*, *1,* 225-260.

Zwiers, J. (1989). *Compositionality, Concurrency and Partial Correctness*. Lecture Notes in Computer Science 321. Berlin: Springer.