Information and Software Technology 1995 **37** (4) 213–224

Static analysis of functional programs

K G van den Berg and P M van den Broek

University of Twente, Faculty of Computer Science, P.O. Box 217, 7500 AE Enschede, the Netherlands
e-mail: vdberg@cs.utwente.nl

In this paper, 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†, 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.

Keywords: functional programming, graph models, software metrics

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[1]. In this paper, 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[2] with respect to the comprehensibility of the programs[3]. 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 is the subject of a separate study[4].

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[5]. They capture the dependencies of objects in the program at different levels of abstraction. For example, 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[6]. Callgraphs for Prolog programs have been given by Fenton and Kaposi[7]. A callgraph model for functional programs in Miranda has been described by Harrison[8]. In this paper, 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[9]. 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 paper.

Flowgraphs are used for the modelling of control-flow in imperative programs[1]. The nodes in the directed graphs correspond to statements in the programs, whereas the edges from one node to the other indicates 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. For example, an elementary action is modelled as flowgraph in Figure 1a (referred to as P_1); the if-then construct in a program is modelled as flowgraph in Figure 1b (referred to as D_0); the if-then-else construct is modelled as flowgraph in Figure 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 1c, is given in Figure 1d. This is denoted as $D_1(D_0)$, in which

*Miranda is a trademark of Software Research Limited.
†Prometrix is a product of Infometrix Software.

SSDI 0950-5849(94)00456-0

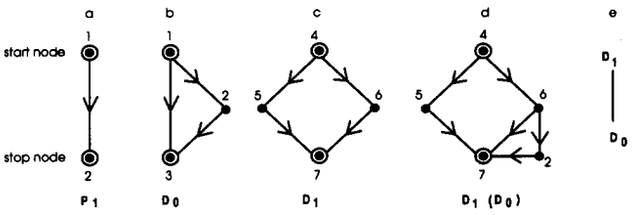

Figure 1 Elementary flowgraphs and decomposition tree

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 1d is depicted in Figure 1e.

In order to quantify internal attributes of software, metrics have been defined on flowgraphs, decomposition trees and callgraphs[1]. 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 paper.

The paper is organized 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.

Functional programs

In this section, some characteristics of programs in the functional language Miranda[2,9] will be described with an example program.

Example program

In Figure 2, 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 (real or integer) 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: 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 expression main 1 4 test evaluates to the string "13 + 5 i".

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

```
| | 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
                                                                    49
```

Figure 2 Example Miranda program

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.

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 Figure 3).

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).

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 in Peyton Jones[10]. Other aspects of the control-flow in the actual evaluation of expressions, such as laziness[9], will be abstracted from.

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 Figure 3) 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, can the clauses 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. For example, 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 can the cases 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*. If there is no other clause there will be a program error.

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

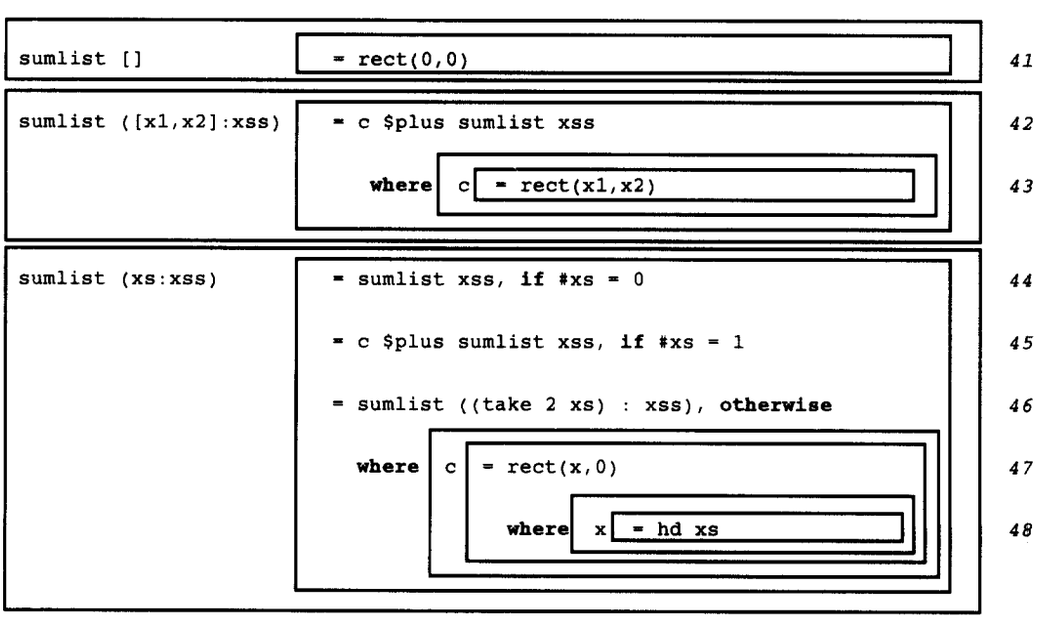

Figure 3 Structure of the definition sumlist

In Miranda, commonly 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 (e.g. lines 27, 28, 41 and 42);
- patterns with the list-constructor: (e.g. xs:xss in line 44);
- patterns with the + operator (e.g. n+1 where n is an integer);
- 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 abstract from the actual content of these expressions, which may be 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.

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 4. 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[1]. 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 1e).

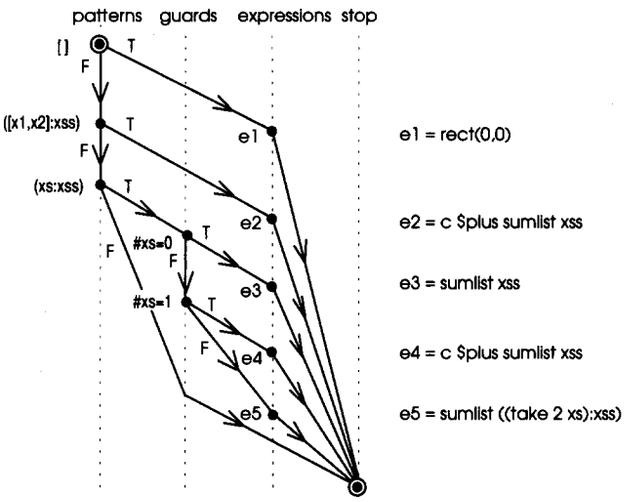

Figure 4 Annotated control-flow graph of the function sumlist

There are simple function definitions resulting in flowgraphs that are not D-structured (i.e. containing other than D_0, D_1, D_2, D_3, and P_1-primes). Consider for example the following function f (the function funnyLastElt[10]):

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 5a.

The decomposition of this flowgraph is $X_1(D_1(D_0))$, where X_1 is the prime given in Figure 5b. The same prime is associated with a lazy boolean and-expression in a selection[12].

Furthermore, from this example it can be shown that guards interact with pattern matching and the order of the clauses. There are six 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':

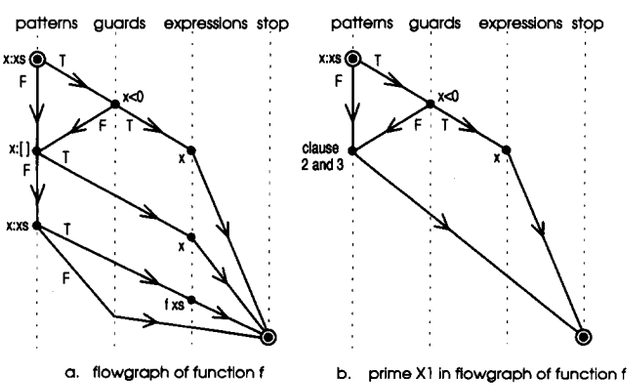

Figure 5 Annotated control-flow graph of the function f with prime X_1

```
f' (x:xs) = x     , 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.

Whether this alternative definition, with a D-structured flowgraph decomposition, should be preferred to the first definition with the X-prime in its flowgraph decomposition, e.g. with respect to the external attribute comprehensibility, has to be established in a separate validation study.

Flowgraph metrics

There are a large number of metrics defined on flowgraphs and decomposition trees[1]. A selection of flowgraph metrics for the function sumlist is given in Table 1. 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[1]. In tools, like Qualms[13] and Prometrix[14], the prime decomposition is used in the computation of the testability metrics.

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 1. 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.

Table 1. Flowgraph metrics for the function sumlist

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

The testability metrics give the number of test cases required in each of the testing strategies. For example, branch testing requires that each edge in the flowgraph be visited at least once; for the function sumlist a minimum of six test cases is required. Statement testing requires that each node in the flowgraph be visited at least once. The test cases can be derived directly from the flowgraph (see Table 2). 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.

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.

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;
- library functions: functions defined in another script or in the standard library;

Table 2. Test cases for the function sumlist

			Patterns and guards				
Test	**Expression**	**Line**	**[ ]**	**[x1,x2]: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	false
5	sumlist ((take 2 xs):xss)	46	false	false	true	false	false
6	—	—	false	false	false	—	—

- primitive functions (or operators): these are in Miranda* the arithmetic operators (+, −, /, *, ^, div, mod), the boolean operators (&, ∨, ~, =, >, <), the list operators (#, :, ++, −−, !), and the function composition operator (.).

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[14]. 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 Figure 2, 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: they 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).

Each of these classes of callgraphs will be discussed in turn.

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 6), the global, local and library functions are:

*See the Miranda manual.

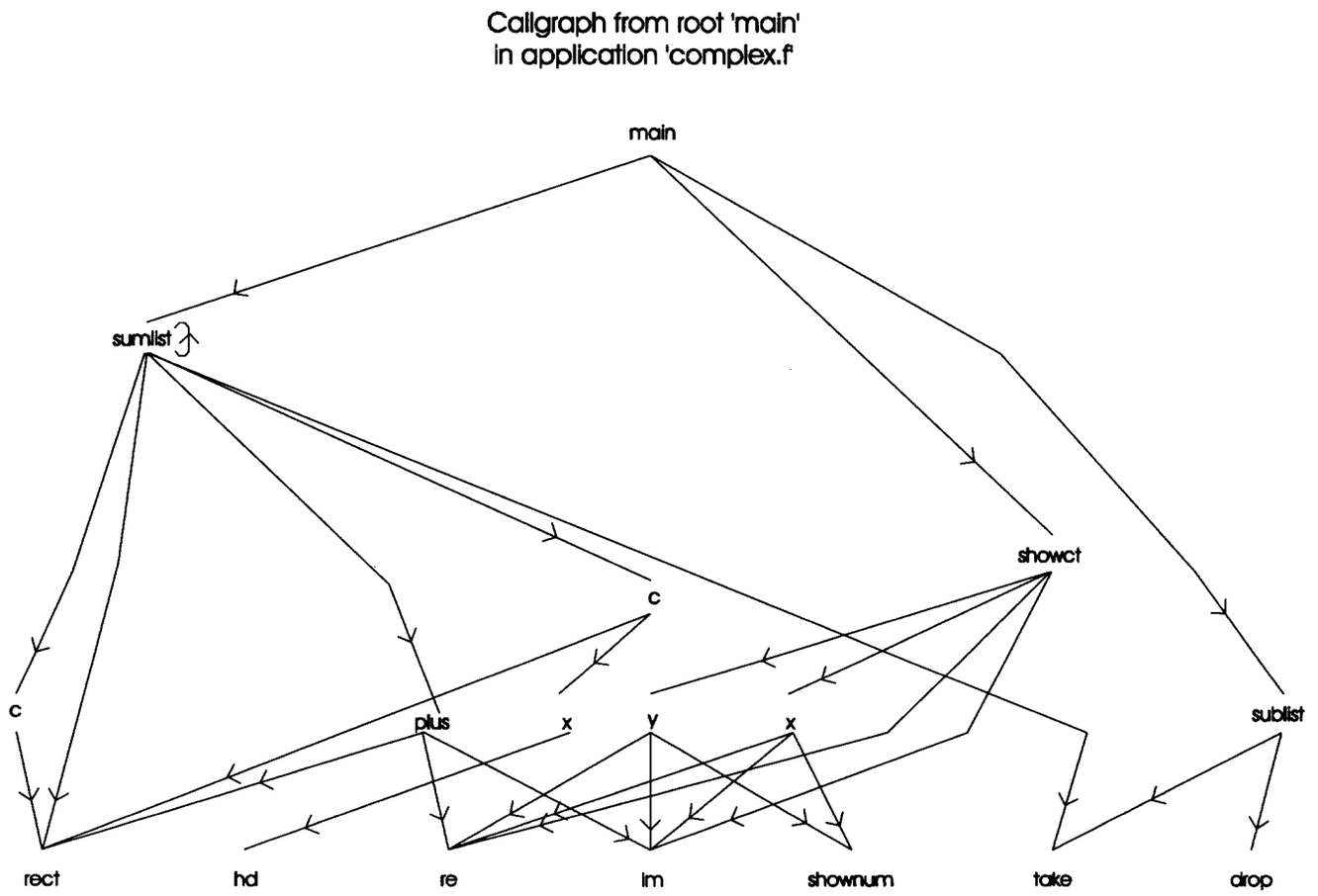

Figure 6 The general callgraph from root main

- 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 6, the library functions are shown.

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 global and library 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 7, the library functions are not shown.

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 localization of the clause in which the local function has been defined. In the local callgraph with as root the function sumlist (see Figure 8), 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 8, the library functions are shown. The edge from \sumlist to sumlist implies a recursive call of the top level function in the root.

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.

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[15]). One abstracts from the actual calls to functions in the included script. The example program (given in Figure 2) could be divided into four scripts (see Figure 10): the file compbas.m with the base operations on complex numbers (line 11–32); the file

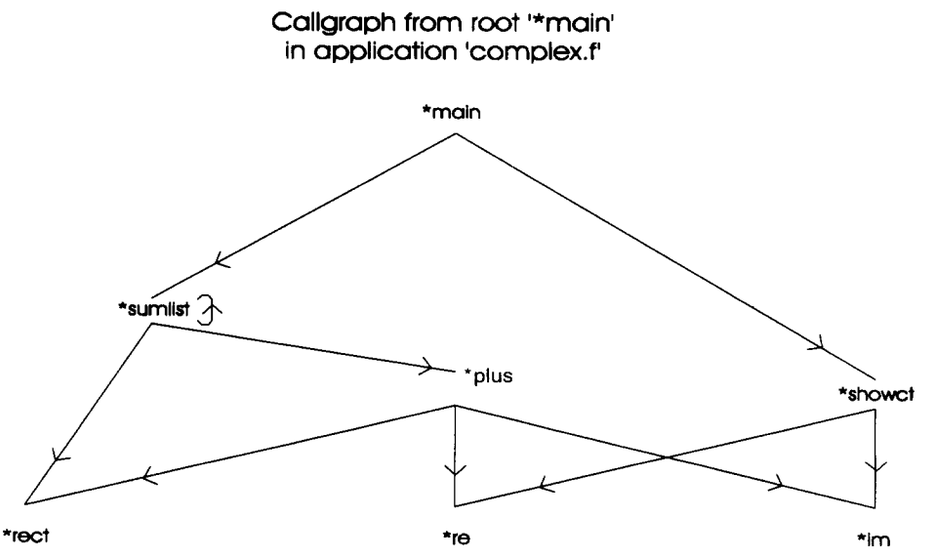

Figure 7 The global callgraph from root main

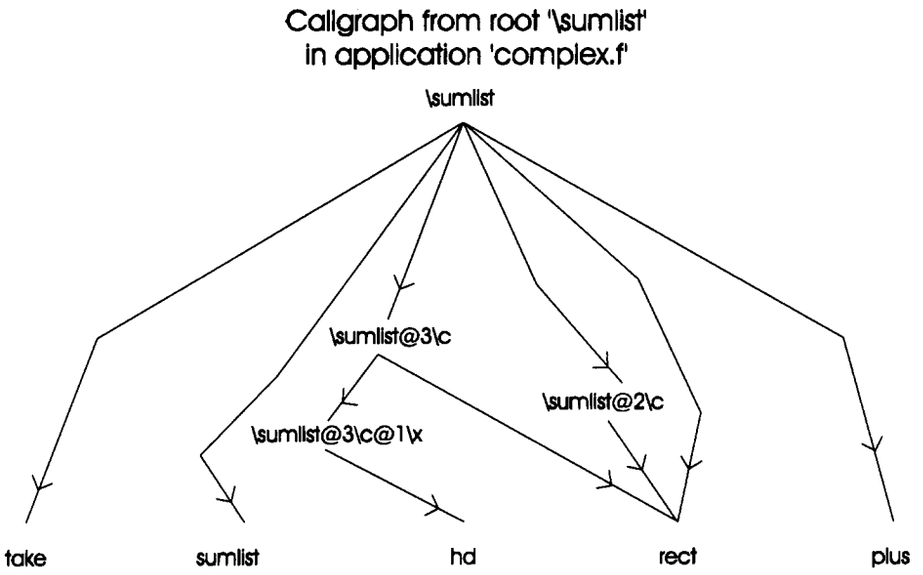

Figure 8 The local callgraph from root sumlist

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 9. The four edges correspond to the four include-constructs in the scripts.

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 3 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.

For these classes of functions, 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;

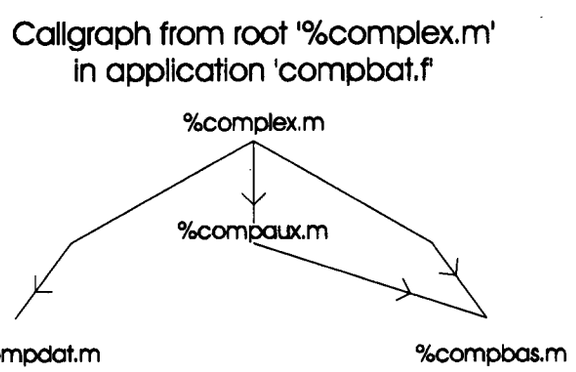

Figure 9 The include callgraph from root complex.m

```
| | 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
```

Figure 10 Example Miranda program with include files

Table 3. Number of functions in callgraphs from root main and root sumlist

Rooted callgraph	# 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

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 4 the number of the function calls, including recursive calls†, 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.

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[1,14]. For each class of callgraphs introduced in the previous section, these metrics are applicable. For the general callgraph with as root the function main, some of these metrics are given in Table 5. The definitions of the metrics are given in Fenton[1]. A short description will be given below.

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 function 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. 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*. 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 normalized 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. For example, 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.

Miranda analyser

In previous research, an analyser has been constructed to obtain the Halstead and McCabe-metrics of Pascal programs and Miranda scripts[16]. The implementation of this analyser was based on an attributed grammar in the synthesizer generator[17]. 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[14] 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.

Prometrix

Prometrix[18] 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.
- The inspect mode: both flowgraphs and callgraphs (from

†In Prometrix recursive calls are not counted.

*In Prometrix 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.

Table 4. Number of function calls for callgraphs from root main and 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. Metrics of the general callgraph from root main

Metric		Value
Size metrics		
—	Number of functions*	17
—	Number of function–function* 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

*In the Miranda analyser (see section 0) the functions are referred to as modules.

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 11, the flowgraph of the function sumlist (cf. the annotated flowgraph in Figure 4).

- 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[18].

Miranda front end

The two main components of the front end are a pre-processor, and an 'editor' generated with the Synthesizer generator[18]. The pre-processor has mainly the following functionality:

- to add semicolons to account for the offside layout rule in Miranda[18];

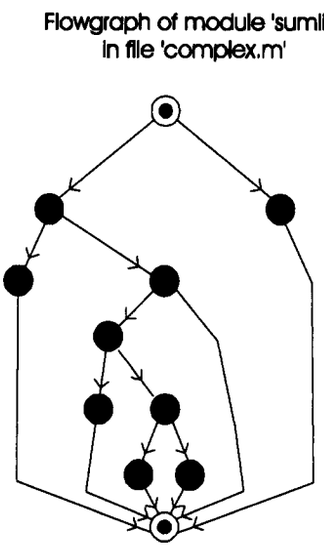

Figure 11 Flowgraph of the function sumlist from Miranda analyser

- to convert 'literate' Miranda scripts to 'normal' scripts[18];
- 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 provide 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†.

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[18]. Figures 6, 7, 8, 9 and 11 are examples of the output from the Miranda analyser.

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 (Figure 2, without include files) is given in Table 6. The values of the maximum, minimum, mean, standard deviation and median for various metrics are given. 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 (Figure 2) is 14 (thus excluding the library functions); the total number of the nodes in the flowgraphs of these functions is 43. Table 5 and 6 are examples of the output

Table 6. Summary statistics of complex.m (without include files)

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	—*	—*	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

*This metric is on an ordinal scale, so this statistical quantity is not appropriate.

†Aliases (see Miranda manual) are not taken into account.

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.

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[19] as executable specifications[20] and for prototyping[21]. 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[22].

Pseudocode

The Miranda metric analyser described in the previous section can be used for designs with stepwise refinement in pseudocode as described below. The design callgraphs obtained in this way gives the 'uses'-hierarchy as in structure charts[3].

In the pseudocode, any Miranda language construct can be used, including the use of local definitions. However, the code needs 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 will be used again to illustrate the design of a Miranda program with this pseudocode. Two steps of refinement are given in Figure 12.

```
|| 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
```

Figure 12 Design of example program in pseudocode

Design callgraph

The design given above can be depicted in a structure chart (see Figure 13) 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 7, but now with the dependencies given in the structure chart of the design. The standard metrics defined on callgraphs can be obtained for these design callgraphs as well.

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 Figure 2.) Similar operations in the graph can be grouped together in one file, e.g. the operations on complex numbers. A modular structure[24] will be obtained such as given in Figure 9.

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 Figure 2, 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,

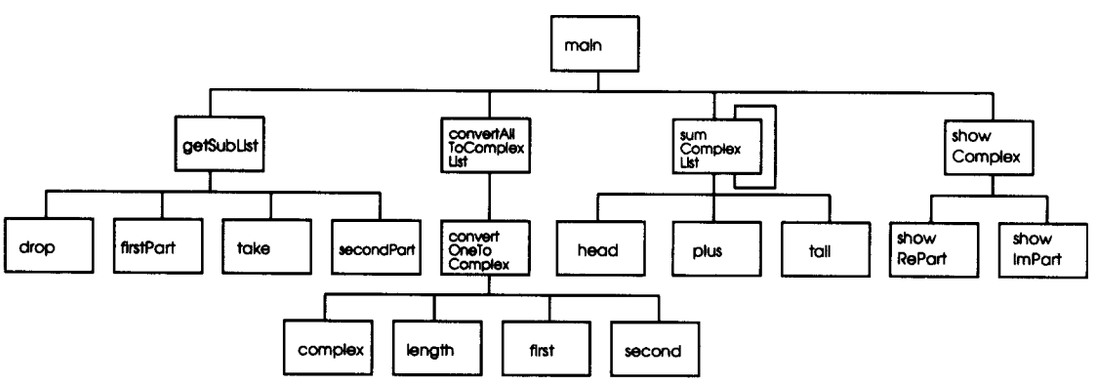

Figure 13 Structure chart of design of example program

resulting in a slightly more efficient program than the one obtained from the design above.

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[4].

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 functions. 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 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[8] showed a model that deviates from our modelling. 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[22] 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 four data files. (The program is roughly equivalent to about 8000 lines of imperative code[23]). 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[4,25,26]. Furthermore, the analyser could easily be extended with a dependency graph of types that are defined in scripts.

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 preset threshold values, e.g. size and testability.

Acknowledgement

The authors would like to thank Richard Bache for providing his notes on building front ends, Axel Belinfante for his advice on the use of the Synthesizer generator, Bert Helthuis for his support in coupling the Miranda front end to Prometrix, Mark Ramaer for the implementation of a part of the pre-processor, and Dick van der Sar for his work on the dependency model.

References

1 Fenton, N E *Software metrics: a rigorous approach* Chapman & Hall (1991)
2 Turner, D A 'An overview of Miranda' *Sigplan Notices* Vol 21 No 12 (1986) pp 158–166
3 Davies, S P 'Models and theories of programming strategy' *Int. J. Man-Machine Studies* Vol 39 (1993) pp 237–267
4 Berg, K G van den and Broek, P M van den 'Programmers performance on structured versus nonstructured function definitions' *Memoranda Informatica* Enschede, University Twente (1995)
5 Yourdon, E and Constantine, L L *Structured design: fundamentals of a discipline of computer program and systems design* Prentice-Hall (1979)
6 Bache, R M *Graph models of software*. PhD dissertation, London, Polytechnic of the Southbank (1990)
7 Fenton, N E and Kaposi, A A 'An engineering theory of structure and measurement' in Kitchenham, B A and Littlewood, B (eds) *Measurement for software control and assurance* Elsevier (1989) pp 335–384
8 Harrison, R 'Quantifying internal attributes of functional programs' *Inf. and Soft. Technol.* Vol 35 No 10 (1993) pp 554–560
9 Bird, R and Wadler, P *Introduction to functional programming* Prentice-Hall (1988)
10 Peyton Jones, S L *The implementation of functional programming languages* Prentice-Hall (1987)
11 Shepperd, M and Ince, D *Derivation and validation of software metrics* Clarendon Press (1993)
12 Fenton, N E and Kaposi, A A 'Metrics and software structure' *Inf. and Soft. Technol.* Vol 29 No 6 (1987) pp 301–320
13 Wilson, L and Leelasena, L *The Qualms program documentation* Alvey Project SE/69, London, South Bank Polytechnic (1988)
14 Prometrix *User and installation manual*, Glasgow, Infometrix Software (1993)
15 Pomberger, G *Software engineering and Modula-2* Prentice-Hall (1984)
16 Berg, K G van den 'Syntactic complexity metrics and the readability of programs in a functional computer language' in Engel, F L *et al.* (eds), *Cognitive modelling and interactive environments in language learning* NATO Advanced Science Institute Series, Berlin, Springer (1992) pp 199–206
17 Reps, T W and Teitelbaum, T *The synthesizer generator* Springer (1989)
18 GrammaTech *The synthesizer generator reference manual* Release 4.1 GrammaTech (1993)
19 Joosten, S M M *The use of functional programming in software development* PhD dissertation, Enschede, University of Twente (1989)
20 Turner, D A 'Functional programs as executable specifications' in Hoare, C A R and Shepherdson, J C (eds) *Mathematical logic and programming languages* Prentice-Hall (1985) pp 29–54
21 Henderson, P *Functional programming, formal specification, and rapid prototyping* IEEE-SE, Vol 12 No 2 (1986) pp 241–250
22 Harrison, R 'A declarative development technique' *Software Quality Management Conference, Southampton* (1993) pp 431–444
23 Turner, D 'Recursion equations as a programming language' in Darlington, J (ed) *Functional programming and its applications* Cambridge University Press (1982) pp 1–28
24 Parnas, D L 'On the criteria to be used in decomposing systems into modules' *Communications ACM* Vol 14 No 1 (1972) pp 1053–1058
25 Berg, K G van den and Broek, P M van den 'Axiomatic testing of structure metrics' *Proc. 2nd Int. Software Metrics Symposium* London, IEEE Computer Society Press (1994) pp 45–53
26 Berg, K G van den, Broek, P M van den and Petersen, G M van 'Validation of structure metrics: a case study' *Proc. Int. Software Metrics Symposium METRICS 93*, Washington, IEEE Computer Society Press (1993) pp 92–99