Chapter 1

Data Flow Considerations for Source Code Directed Testing of Functional Programs

Manfred Widera[1]

Abstract: Testing of software components during their development is a heavily used approach to detect programming errors and to evaluate the quality of software. Systematic approaches to software testing get a more and more increasing impact on software development processes.
For imperative programs there are several approaches to measure the appropriateness of a set of test cases for a program part under testing. Some of them are source code directed and are given as coverage criteria on flow graphs.
In this paper data flow analysis is identified as an important tool for source code directed testing of functional programs. A powerful adaption of data flow analysis to the needs of source code directed testing is defined, and two main applications are discussed: the calculation of possible destinations of function calls, and the coverage analysis for test cases.

1.1 INTRODUCTION

Testing of software is a widely used method of detecting errors during the software development process. One can assume every software to be tested before being put to use in practice. Though testing can just prove the presence but not the absence of errors, the passing of all tests given by an appropriate test set is often understood as an evidence for reaching a certain level of software quality. For imperative programming there are several approaches defining the appropriateness of a test set by coverage criteria based on the flow graph of the tested

[1]Fachbereich Informatik, FernUniversität in Hagen, D-58084 Hagen, Germany, Email: Manfred.Widera@fernuni-hagen.de

program. Testing in this way is called source code directed or structure directed, and is usually applied to small program fractions like single modules.

In the context of functional programming, there just exist simple ad hoc approaches to source code directed testing, as e.g. the cover tool for Erlang [NN03] that checks the individual *lines* of a program for coverage. As systematic testing is an important task of professional software development, it is desirable to have more advanced source code oriented testing methods for functional programming languages available.

For functional languages data flow analysis plays an important role in the context of source code directed testing. A definition of data flow analysis is given, that is applicable to source code directed testing for the functional language Erlang [AVWW96]. Two main application areas of data flow analysis during the testing process are identified. First, data flow analysis is necessary during the flow graph generation for higher order programs. It provides the set of destinations, a function call can point to. Second, several coverage criteria based on the data flow are known for imperative languages. These criteria for data flow directed testing seem to be especially promising for functional languages.

We further discuss that source code directed testing can be done completely based on the data flow without the need to consider the control flow. This makes it especially useful for functional programming languages with lazy evaluation.

The rest of the paper is organized as follows. Section 1.2 discusses some work from the literature which is related to the results in this paper. In Sec. 1.3 some preliminaries are given by briefly describing the standard testing cycle, and the subset of the Erlang language considered in the rest of this paper. Section 1.4 describes the top level structure of the testing system and briefly describes its three main components. In Sec. 1.5 the basic notions for data flow analysis are defined, and the use of data flow analysis during the flow graph generation as well as for the coverage analysis are discussed. Section 1.6 discusses how the concept of source code oriented testing can be transfered to other functional languages, especially those employing lazy evaluation. Section 1.7 finally gives a conclusion and some notes on future work.

1.2 RELATED WORK

Flow graphs in functional programming, and their use for testing functional programs are related to approaches from several areas. There are already approaches on flow graphs for functional languages. Van den Berg [vdB95] uses flow graphs and call graphs in the context of software measurement for functional programs. The flow graphs used there consider function calls as atomic operations and are generated for each function independently. Information on calls between functions is given by a call graph as separate structure.

The concepts of generating flow graphs for higher order programs is described by Shivers [Shi88] where several different levels of precision for the needed data flow analysis are proposed. These different precision levels are further analyzed by Ashley/Dybvig [AD98]. Especially, the level 0CFA is similar to our approach.

Due to its use of continuation passing style (CPS) and the Y combinator, it is, however, not very adequate for presenting the analysis results to human programmers. The same holds for works based on Shivers approach [Shi88]. They do not focus on the presentation of the generated flow graphs to the programmer.

Different approaches on testing and debugging functional programs have been proposed. QuickCheck [CH00] aims at automatically checking Haskell programs by generating input data on a random basis and checking the results with constraints on the expected output. In the WYSIWYT framework [RBL^{+}01, RLDB98, RCB^{+}00] flow graphs are used for judging the coverage of a functional program by a set of test inputs. This approach is, however, restricted to spreadsheets considered as first order functional programs without recursion.

Several approaches on declarative debugging and tracing functional languages (e.g. [Gil00], [Nai97], [Chi01, WCBR01]) describe how to trace down the programming errors causing an observed misbehavior of a program. These approaches, however, do not provide mechanisms for generating or judging the test sets that are used to provoke such a misbehavior.

The module cover that comes with the tools library of Erlang [NN03] implements a coverage test for Erlang source code that analyzes the individual lines of the source code for coverage. It does not use an interpreter, but compiles the modules to be analyzed in a special way. Cover does not need to employ data flow analysis, since it directly works on the program source code. As a drawback, it is, however, not able to distinguish several computations coded within a single line, or to check non-local relationships, e.g. between calls and called functions, between throws and corresponding catches, or between definitions and reached uses of values.

1.3 PRELIMINARIES

1.3.1 An Overview over the Testing of Software

Literature on testing imperative programs (e.g. [Lig02]) usually distinguishes three different stages of testing during the software development process.

1. Component testing
2. Integration testing
3. System testing

Component testing focuses on testing the single units of a software product *early* in the development process. Source code directed coverage criteria are used in this stage to ensure that every program part is executed at least once. (The definition of the term *program part* that applies in a certain situation is given by the chosen coverage criterion.)

For *integration testing* several units (that have been tested individually during component testing) are connected to each other, and their interaction is tested.

This stage is especially necessary to check the correct implementation of interfaces by connected components.

In the *system testing* phase the overall software product is tested. Among others, this stage contains e.g. stress tests, and tests by selected users with realistic applications.

For the remainder of this paper, it is important to note that source code directed testing methods only make sense in the context of component testing. Therefore, the code fragments and test cases, source code directed testing has to deal with, are rather small and elementary.

1.3.2 Syntax of the Considered Erlang Subset

Though many of the considerations on structure oriented testing apply to different functional programming languages, we focus on the programming language Erlang [AVWW96] in the following. Some notes on transferring the concept to other languages are given in Sec. 1.6.

In order to simplify the presentation, we restrict the considered Erlang subset to Fig. 1.1, that essentially covers the sequential part of Erlang without some syntactic sugar. Definitions marked by a $\star$ are not of interest for source code directed testing, and are therefore omitted here. Infix operators are considered as ordinary functions. Due to the importance of the BIF (built in function) *throw* for the control flow, it has the state of a syntactic keyword in this work. In the following when speaking of a first order function call we mean a call of the form $fn(e_1, e_2, \ldots, e_k)$ with a function name fn, and a higher order function call has the form $e_0(e_1, e_2, \ldots, e_k)$.

1.4 STRUCTURE OF THE TEST ANALYZER

The top level structure of our test analysis system is given by a sequence of the following three stages.

1. Flow graph generation.
2. Supervised evaluation of one or several tests in the flow graph.
3. Analysis of the test set according to a specified coverage criterion.

The following subsections contain an overview over these three stages with special focus on the data flow considerations necessary for the respective stage.

1.4.1 Flow Graph Generation

The generation of flow graphs is described in other publications [Wid03, Wid04b] and is therefore just sketched here. Given an Erlang module or a list of modules, the flow graph generation consists of the following four steps.

1. The modules are read and preprocessed to meet the following properties.

constants a: $\star$
variables X: $\star$
patterns p: $a|X|\{p_1,\ldots,p_k\}|[p_1|p_2]|[p_1,\ldots,p_k]$
guards g: $\star$
if clauses ic: $g \rightarrow l$
case clauses cc: p [when g] $\rightarrow l$
fun clauses fc: $(p_1,\ldots,p_k)$ [when g] $\rightarrow l$
function name fn: $\star$
expressions e: $a|X|e_0(e_1,e_2,\ldots,e_k)|fn(e_1,e_2,\ldots,e_k)|$
$\{e_1,\ldots,e_k\}|[e_1|e_2]|[e_1,\ldots,e_k]|$begin l end$|$
if $ic_1;ic_2;\ldots ic_k$ end$|$case e of $cc_1;cc_2;\ldots;cc_k$ end$|$
fun $fc_1;fc_2;\ldots;fc_n$ end$|$catch $e|$throw e
expression lists l: $e_1,e_2,\ldots,e_k$
functions f: $fn\,fc_1;fn\,fc_2;\ldots;fn\,fc_n$.
programs P: $f_1 f_2 \ldots f_k$

FIGURE 1.1. The Erlang subset under consideration

- Every function has unique entry and exit points.
- Each function has an *import node* indicating the local definition of the formal parameters, a *context node* indicating the local definition of the variables taken from the context of the function definition, and a *return node* denoting the leaving of the function's code, and the returning of the value bound to the return variable.
- All arguments to operations (function calls, structure generation, ...) are variables.
- The return variable containing the return value of the function is bound before reaching the return node.

2. Flow graphs for the individual functions are generated. For each expression a node is generated (especially containing a node number which is unique within the module), and the nodes are connected in a straightforward manner. There exists a number of different node representations for different expressions in the source code [Wid04b].

3. For each function call in the graph the possible destinations of the call are calculated, and the corresponding call edges are introduced. For first order calls the destination is given by the name of the called function; for higher

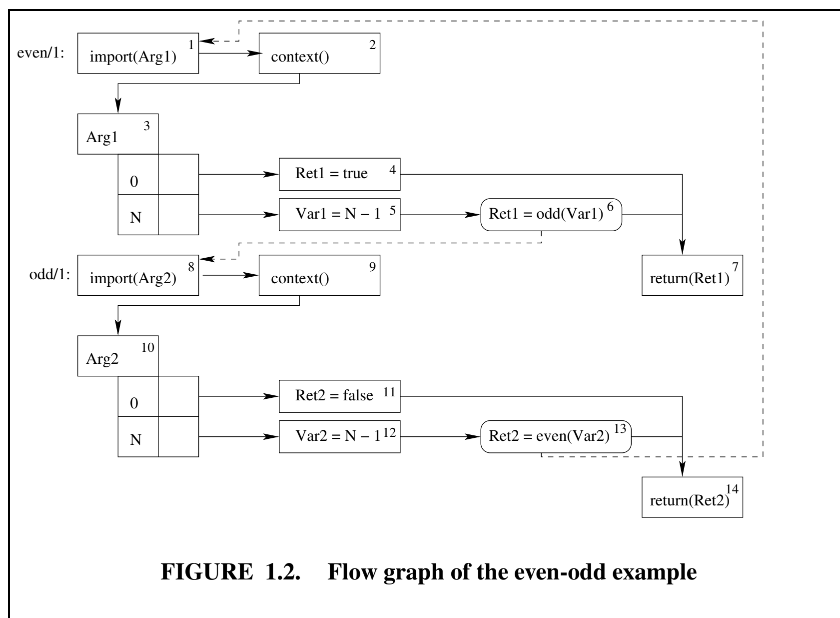

FIGURE 1.2. Flow graph of the even-odd example

order calls, an iterated data flow analysis process is necessary to compute all possible destinations. Call edges represent the control flow during calling a function, *and* the return from the call [Wid04b].

4. For each `throw` expression in the code, indicating a non-local return, the corresponding destinations given by `catch` expressions are calculated and edges are introduced.

Example 1.1. Consider the following definition of the functions even and odd.

```
even(0) -> true;            odd(0) -> false;
even(N) -> odd(N - 1).      odd(N) -> even(N - 1).
```

The flow graph of a module containing exactly these two function definitions is given in Fig. 1.2. Call edges are denoted by dashed lines. The numbers at the right top corners of the nodes are the unique node numbers.

1.4.2 Supervised Evaluation of Test Cases

During the evaluation of the test cases, a trace through the flow graph needs to be generated in order to enable the subsequent analysis according to a coverage criterion. For this reason, some kind of supervised evaluation is necessary.

Example 1.2. Consider the flow graph from Ex. 1.1 and the test `even(1)`. This test yields the result `false` and the trace

```
[1, 2, 3, 5, 6, [8, 9, 10, 11, 14], 7]
```

The sublist denotes the trace of the subcall `odd(0)` occurring during the evaluation.

An interpreter for flow graphs was chosen for performing the supervised tests and for collecting the needed trace. The reasons for preferring an interpreter over some sort of instrumented byte code (by modifying the compiler or by employing a preprocessor) are the simplicity and extendibility of this approach, its superior potential for extensions towards concurrent Erlang, and its reusability for a partial evaluation stage necessary during the flow graph generation [Wid04a].

1.4.3 Judging the Test Coverage

Given the flow graph and the execution trace from the interpretation of some test cases, we can calculate the parts of the flow graph that were already covered by the tests. Several coverage criteria are known for imperative programming languages [ZHM97]. For functional programming, we expect two kinds of coverage criteria to be of special use.

- The *node coverage criterion* and the *edge coverage criterion* express a basis of coverage that should be met by every test set. They give moderate extensions of the line coverage criterion tested by the Erlang tool *cover* [NN03].
- Data flow oriented definition-use criteria reflect the declarative character of functional programming that tries to abstract from a certain control flow. As a further advantage, we suppose definition-use criteria to carry over to functional programming languages with lazy evaluation like Haskell [Jon03], and to the concurrent programming constructs of Erlang easily.

Example 1.3. Consider the flow graph and the trace of Ex. 1.2. Judging this trace according to the node coverage criterion, the nodes 1, 2, 3, 5, 6, 7 ,8, 9, 10, 11, and 14 are covered by the test; the nodes remaining untested are 4, 12, and 13.

1.5 STRUCTURE ORIENTED TESTING AND DATA FLOW ANALYSIS

Data flow analysis is one of the main tools necessary for structure oriented testing. Especially, data flow analysis is employed during the following two steps.

- For higher order function calls the possible destinations of a call depend on the data flow of the definitions reaching the use in the function position of the call. Therefore, data flow analysis is necessary for calculating the call edges of higher order calls.
- Data flow analysis is the basis for calculating the definition-use pairs whose coverage is checked by the definition-use coverage criteria.

In the rest of this section the definition of some basic notions for data flow analysis is given before we discuss the two uses of data flow analysis mentioned above in more detail.

1.5.1 Basic Notions of Data Flow Analysis

As a basis for the remaining discussion, we give a definition of some notions for data flow analysis in this section. The definitions given here are known for imperative languages, but need some adaption towards the specialties of Erlang (e.g. the pattern matching), and their intended use (e.g. an extensive definition of a definition reaching a use). For a definition of a variable v we write $def(v)$, and for a use of v we write $use(v)$. Their precise definitions are as follows.

Definition 1.1 (Definitions). *Let G be a flow graph, and v a variable. A node n in G contains a definition of v if one of the following conditions holds:*

- *n is an import node and v is one of the variables defined in n.*
- *n is a context node and v is one of the variables defined in n. This is called f-definition (denoted by f-def(v)).*
- *n denotes a pattern matching `LHS = RHS`, v occurs in `LHS`, and there is at least one path w from the beginning of the function containing n to n itself such that v is not defined on w.*
- *n denotes a conditional, v occurs in at least one pattern p in n, and there is at least one path w from the beginning of the function containing n to n itself such that v is not defined on w.*

A definition binding v to a value selected from a structure (either by pattern matching or the corresponding selection BIFs) is called an s-definition and denoted by s-def(v).

The opposite of the definition of a variable is given by its use. Intuitively, a use of a variable v is given by every expression that needs the value of v to be evaluated.

Definition 1.2 (Uses). *Let G be a flow graph, and v a variable. A node n in G contains a use of v if one of the following conditions holds:*

- *n is a node representing the expression E or a match $p = E$ where*
  - $E = v$
  - $E = \{v_1, \ldots, v_k\}$, $E = [v_1|v_2]$, *or* $E = [v_1, \ldots, v_k]$ *with* $v = v_i$ *for some i. This is called an s-use and denoted by s-use(v).*
  - $E = v_0(v_1, \ldots, v_k)$ *or* $E = fn(v_1, \ldots, v_k)$ *with* $v = v_i$ *for some* $i \in \{0, \ldots, k\}$.
- *n is a conditional node with a test given by v.*
- *n contains the generation of a fun, and there is at least one path w from the beginning of the function containing n to n itself such that v is defined on w. This is called an f-use and denoted by f-use(v).*

- *n denotes a pattern matching* `LHS = RHS`*, v occurs in* `LHS`*, and there is at least one path w from the beginning of the function containing n to n itself such that v is defined on w.*
- *n denotes a conditional, v occurs in at least one pattern p in n, and there is at least one path w from the beginning of the function containing n to n itself such that v is defined on w.*

The above definitions show one of the main specialties of pattern matching: a pattern match containing a variable X can be either a definition or a use of X (or even both), depending on previous definitions of X on the possible paths to the pattern match.

Some special information is added to the specification of a definition or use of a variable v inside a pattern p of a conditional. Besides the node n of the conditional it contains the number of the clause, the pattern p belongs to. Occurrences of v in the patterns of several clauses of n are treated independently.

For both, f-definitions and f-uses as well as s-definitions and s-uses, we need to define the notion of *corresponding* uses and definitions.

An *f-use* is the use of a variable that freezes the variable in a function closure. A corresponding *f-def* defrosts exactly this value in a local definition.

Definition 1.3 (corresponding *f-use*, *f-def*). *Let v be a variable, u an f-use of v, and d an f-definition of v. u and d correspond to each other if the fun containing d is the one defined in u.*

An *s-use* uses a variable to store it in a structure. A corresponding *s-def* selects exactly this value at the right structure position from the right structure value.

Definition 1.4 (corresponding *s-use*, *s-def*). *Let v be a variable, and u an s-use of v, generating a structure c. A selection d defining a variable v' is an s-definition of v' corresponding to u if the structure decomposed in d is c, and the selected element position is the one containing the value of v.*

Note that for an s-use and the corresponding s-def the variable names can differ.

The following main definition of this section states the situations under which a definition d reaches a use u.

Definition 1.5. *Let d be a definition of a variable v, and u a use of a variable v'. Then d reaches u if one of the following properties holds:*

- *$v = v'$ and there is a path in the flow graph from d to u that does not contain a definition of v different from d. In this case we say d reaches u* directly.[2]
- *There is a copy expression e of the form $\tilde{v} = \tilde{v}'$ such that d reaches the use of $\tilde{v}'$ in e and the definition of $\tilde{v}$ in e reaches u.*
- *d reaches an f-use of some $\tilde{v}$ and there is a corresponding f-definition of $\tilde{v}$ that reaches u.*

[2]Function calls need to set all variables to undefined implicitly, before transferring the control to the function body.

- *d reaches an s-use of some $\tilde{v}'$ and there is a corresponding s-definition of some $\tilde{v}$ that reaches u.*

Note that each rubber band edge from a function call c to a function f/k contains implicit assignments $p_i = a_i$ for each $i = 1,\ldots,k$ where p_i is the ith parameter in the definition of f and a_i is the ith argument in the call c. For the return an additional assignment of the return variable of f to the variable assigned to c is given. These implicit assignments are processed like ordinary renamings according to Def. 1.5.

Example 1.4. Consider the flow graph of the even-odd example discussed in Ex. 1.1 and presented in Fig. 1.2. Consider the definition of the value *Var1* in Node 5. This value is used in Node 6, and by the implicit copy expression given by the call edge, the definition of *Arg2* in Node 8 gets an alias of it. *Arg2* is used in Node 10, and *N* defined in the second pattern becomes an alias of *Arg2*. *N* is used in Node 12, and because of the aliasing the definition of *Var1* in Node 5 reaches the use of *N* in Node 12.

1.5.2 Data Flow Analysis in the Flow Graph Generation

Recall the four steps of flow graph generation:

1. Reading of the module(s) to be transformed into a flow graph.

2. Generation of flow graphs for the individual functions.

3. Calculation and insertion of call edges.

4. Calculation and insertion of throw edges.

During this process, data flow analysis is needed during the last two steps that calculate relations between distant program parts. The data flow analysis process can, however, be supported by collecting some local data flow information during the generation of the function flow graphs in Step (2). Therefore, the following local data flow information is stored in the flow graph.

- For every function, information on the values it potentially returns and throws are collected. It is local information in the sense, that returns and throws from other functions are represented by abstract values (which e.g. denote the return value of the call in node *n*).
  In functions containing a catch, the values thrown by its subexpression are correctly processed in the functions return value.

- For every node containing a use of a value, the used values and the corresponding reaching definitions are stored. This information is local in the sense that the nodes containing reaching definitions belong to the same function. Definitions coming from outside are shadowed by the import node of the function, the context node, or some call node of the function.

Using this information, call edges can be calculated in an iterated process, similar to 0CFA [Shi88], that works as follows.

1. For each first order call there is a unique destination function. If it belongs to the flow graph, a call edge is inserted.
2. For each higher order call, the set of definitions reaching the use in the function position is calculated. This is done taking the needed piece of local information as generated in Step (2), and by replacing abstract values by the corresponding global data, according to the current intermediate state of the flow graph.
   Those definitions that denote a fun generation or a tuple of two atoms[3] give the possible destinations of the call. For each of these destinations, that belongs to the flow graph, a call edge is inserted.
3. Step (2) is repeated until a fixed point is reached, and no new call edges are generated.

1.5.3 Coverage Criteria Based on Data Flow Analysis

The data flow oriented coverage criteria are based on covering a certain subset of the du-pairs (i.e. the pairs (d,u) where d is a definition reaching u) occurring in the tested program.

In the literature for imperative programming the following data flow oriented coverage criteria can be distinguished [ZHM97].

- The *all definitions* criterion requires that for every definition that reaches a use, there is at least one subtrace from the definition to one of its uses.
- The *all uses* criterion is stronger and requires that for all definitions d and all uses u the definition d reaches, there is a subtrace on which d reaches u.
- The *all du-paths* criterion requires for each definition d, each use u reached by d, and each path p from d to u such that d reaches u on p there is a subtrace representing p.

When distinguishing *predicate uses (p-uses)* occurring in the predicate of a conditional and *computation uses (c-uses)* occurring in other computations [ZHM97], several combinations are possible, that especially focus on the p-uses, on the c-uses or on both of them. Because of the structure of Erlang conditionals based on pattern matching, the separation of *p-uses* and *c-uses* seems, however, not appropriate.

Further criteria that are based on a combination of several definitions are not discussed here, because we believe them to be to complex to be of practical use for functional programming.

For each of the mentioned criteria, we can define several different coverage criteria by modifying the notion of a definition reaching a use defined in Def. 1.5.

[3]Partial evaluation is applied to be as accurate as possible in computing the possible destinations.

- The f-defs in a context node can be part of the considered du-pairs, or they can be replaced by the definitions reaching the corresponding f-uses. (Considering corresponding f-uses results in considering relations of values that cross function borders.)

- The considered du-pairs can contain copy expressions as definitions, or the definitions reaching the uses in the copy expressions. (The most interesting applications for copy expressions are the implicit copy expressions given by the copying of call arguments to the formal parameters of the called function.)

- The considered du-pairs can either contain s-defs, or all s-defs are replaced by the definitions reaching the corresponding s-uses. (Detailed tests are necessary here in order to find a good compromise between the expense of performing all necessary tests and the desire to detect as many errors as possible by testing).

1.6 STRUCTURE ORIENTED TESTING FOR OTHER FUNCTIONAL LANGUAGES

The ideas described here, and the structure oriented testing approach based on them, as well as on further works [Wid03, Wid04b, Wid04a] are described with the specialties of Erlang in mind. They, however, also apply to other functional programming languages in general.

When transferring the approach to functional languages based on lazy evaluation, like e.g. Haskell, the different evaluation order needs to be addressed explicitely. This is discussed for the three stages of the testing process

1. flow graph generation,

2. test case execution,

3. evaluation of the coverage criterion

independently. For the transfer to lazy languages it is important to note, that there are no large differences between the data flow of eager languages and lazy languages. Even in the case of a lazy language, every definition is evaluated to a sufficient extend when a corresponding use is reached.

1.6.1 Flow Graph Generation

In general, a Haskell flow graph can be generated as described for Erlang programs. Import, context and return nodes can be introduced in the same way. The introduction of fresh variables for arguments of operations can be done by the *let* syntax available in Haskell and most other functional languages.

The calculation of the call edges also carries over directly from Erlang to Haskell programs. This is the case, because the destinations of call edges only depend on the data flow towards the edge source, not on the control flow.

In processing the exception mechanism of the I/O monad one just has to reflect the fact that the catch operation does not return the caught exception (as the Erlang catch does), but passes it to a dedicated exception handler.

The resulting flow graph does however *not* represent the *control flow* of the program. What makes it useful is its representation of the data flow. A definition just generates a thunk when evaluated, and we can be sure that this thunk is evaluated to a sufficient extend before any use it reaches.

1.6.2 Supervised Test Case Execution

The supervised test case execution must perform two subtasks: it must calculate the correct evaluation result, and it must provide a trace representing the evaluation.

Computing the correct evaluation result is a task that is well understood for lazy languages. We just want to note, that implementing an interpreter becomes much more complex than in the eager evaluation case [Wid04a]. Therefore, executing flow graphs based on the given compilation techniques might be preferable.

In generating traces for languages with lazy evaluation, the control flow becomes much more complex and less easy to understand. It is, however, still the only information, that can be used to judge all coverage criteria.

We propose to enrich the trace by thunk information for lazy languages: each thunk gets a unique label assigned on generation, and the thunk generation is marked with this label. Starting and finishing the thunk evaluation are also marked in the flow graph citing the label. By bracketing the evaluation of a thunk with corresponding entries, insertions of thunk evaluations, that are not of interested for a certain analysis, can be ruled out easily.

1.6.3 Evaluation of Coverage Criteria

The control flow of a program is quite complex in a language with lazy evaluation. Furthermore, it is normally not of great interest and subject to change e.g. in an (especially implicitly) parallel environment performing speculative evaluations [HM99]. Thus, we do not assume the control flow directed coverage criteria to be of much use in the context of lazy evaluation.

For the data flow oriented criteria, the situation is, however, different, since we can be sure that a thunk generated by a definition is always evaluated to a sufficient extend before reaching a use. Data flow oriented criteria can be used in the same manner for lazy evaluation as for languages with eager evaluation, and the flow graph can reflect the du-pairs in a manner that is strongly related to the program's source code, and therefore easily understandable for the programmer.

1.7 CONCLUSION AND FUTURE WORK

Source code directed testing is an approach that is heavily used in industry. It therefore needs to be addressed for functional languages as well, in order to in-

crease their acceptance.

Data flow analysis has been identified as an important tool for the source code directed testing process. We have given a definition of the basic notions for data flow analysis, where especially the notion of a definition reaching a use is quite powerful allowing for aliasing, storage into a structure, and freezing during a lambda closure generation.

Two important applications of data flow analysis have been identified during the source code directed testing process. The flow graph generation depends on data flow analysis, because the possible control and data flow in a program depends on the set of lambda closure definitions that reach a higher order function call as use. For the definition of the different coverage criteria, data flow related coverage based on the coverage of definition-use pairs is one important alternative that was adapted from imperative to functional programming languages.

For transferring the idea of source code directed testing to lazy functional languages it turned out important, that the whole concept can be based on data flow analysis without any need to consider the control flow. Since the data flow does not differ for eager and lazy evaluation (even in lazy evaluation, we can be sure that every definition is evaluated to a sufficient extend when reaching a use), the transfer of source code directed testing with data flow oriented coverage criteria carries over directly.

Future work especially focuses on the use of data flow analysis for the coverage tests. While the flow graph generation and the flow graph interpreter have already been implemented for the sequential part of the language Erlang, data flow oriented test coverage is the next part of the system to be implemented. Several versions of definition-use coverage will be implemented, based on different notions of a definition reaching a use. The resulting implementations will be compared in a case study.

REFERENCES

[AD98] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 20(4):845–868, July 1998.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 2nd edition, 1996.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y., September 18–21 2000. ACM Press.

[Chi01] Olaf Chitil. A semantics for tracing. In *Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL*, 2001.

[Gil00] Andy Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop. Technical report of the University of Nottingham*, 2000.

[HM99] Kevin Hammond and Greg Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, April 2003.

[Lig02] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2002.

[Nai97] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[NN03] NN. *Tools version 2.3*, 2003. Documentation for Erlang OTP R9C.

[RBL+01] Gregg Rothermel, Margaret Burnett, Lixin Li, Christopher DuPuis, and Andrei Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, 2001.

[RCB+00] Karen J. Rothermel, Curtis R. Cook, Margaret M. Burnett, Justin Schonfeld, T. R. G. Green, and Gregg Rothermel. WYSIWYT testing in the spreadsheet paradigm. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239. ACM Press, June 2000.

[RLDB98] Gregg Rothermel, Lixin Li, Christopher DuPuis, and Margaret Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 198–207. IEEE Computer Society Press/ACM Press, 1998.

[Shi88] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.

[vdB95] Klaas van den Berg. *Software Measurement and Functional Programming*. 1995.

[WCBR01] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop, Firenze, Italy*, pages 151–170, 2001.

[Wid03] Manfred Widera. Towards flow graph directed testing of functional programs. In Phil Trinder and Grag Michaelson, editors, *Draft Proceedings of the 15th International Workshop on the Implementation of Functional Languages, IFL*, 2003.

[Wid04a] Manfred Widera. Flow graph interpretation for source code directed testing of functional programs. In Clemens Grelck and Frank Huch, editors, *Implementation an Application of Functional Languages, 16th International Workshop, IFL'04*, Technischer Bericht 0408. Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, 2004.

[Wid04b] Manfred Widera. Flow graphs for testing sequential erlang programs. In *Proceedings of the 3rd ACM SIGPLAN Erlang Workshop*, 2004.

[ZHM97] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.