Flow Graphs for Testing Sequential Erlang Programs

Manfred Widera
Fachbereich Informatik, FernUniversitt Hagen
D-58084 Hagen, Germany
Manfred.Widera@fernuni-hagen.de

Abstract

Testing of software components during 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.
This paper gives a definition of flow graphs for Erlang programs and describes a tool for generating such flow graphs. It provides a first step towards the transfer of advanced source code directed testing methods to functional programming.

Categories and Subject Descriptors: D.2.5 **[Software Engineering]**: Testing and Debugging; D.1.1 **[Programming Techniques]**: Applicative (Functional) Programming

General Terms: Algorithms, Design

Keywords: Flow graphs, test coverage

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. Testing in this way is usually applied to small program fractions like single modules.

Erlang'04, September 22, 2004, Snowbird, Utah, USA.

In the context of Erlang, there just exists one ad hoc approach to source code directed testing that checks the individual *lines* of a program for coverage [4]. As systematic testing is an important task of professional software development, it is desirable to have more advanced source code oriented testing methods for Erlang available.

The aim of this paper is to give a definition of flow graphs of (sequential) Erlang programs similar to the known flow graph definition for imperative programs, and to describe a system generating such a flow graph from a program's source code. This forms the first step towards making the large area of systematic, source code directed testing available for Erlang programs. The usefulness of the flow graph definition is shown by carrying over the definition of some basic coverage criteria without much effort.

The rest of the paper is organized as follows. Section 2 gives an overview over related work. In Sec. 3 a restriction of the language under consideration is given and a preprocessing task for programs is sketched. The definition of flow graphs is given in Sec. 4. Section 5 contains the necessary definitions for data flow analysis that is needed for generating flow graphs of higher order programs. The generation of the flow graphs is sketched in Sec. 6, and Sec. 7 gives an outlook on the use of flow graphs for test case analysis. Some conclusions are given in Sec. 8.

2 Related Work

The work presented here is related to publications from several areas. There are already approaches on flow graphs for functional languages. Van den Berg [13] 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 [12] and further analyzed by Ashley/Dybvig [1]. Especially, the level 0CFA described there is very 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 [12]. 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 [3] 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 [9, 10, 11] 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. [6], [8], [2, 14]) 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 [4] implements a coverage test for Erlang source code that analyzes the individual lines of the source code for coverage. 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 or between throws and corresponding catches.

3 Preliminaries

For presenting the definition and generation of functional flow graphs we do not consider the whole Erlang language. We rather concentrate on the subset defined in Fig. 1 (ignoring the boxes around some expressions for the moment). Definitions consisting of a $\star$ are not needed here, and are therefore omitted. Infix operators are considered as ordinary functions. Due to the importance of the BIF *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 $e_0(e_1, e_2, \ldots, e_k)$.

Programs in the considered Erlang subset are normalized to meet the following properties: every function just consists of *one* clause with only variables as arguments, and all function calls, and language constructs just have variables as arguments.

The first condition is met by introducing a new form of expression, generated by the keyword *funcase*, of the form

$$\text{funcase } (\boxed{e_1}, \ldots, \boxed{e_k}) \text{ of } fc_1; fc_2; \ldots; fc_k \text{ end}$$

which expresses the branching given by the original clauses.

To meet the second condition, we replace all expressions that are boxed in Fig. 1, and in the new *funcase* construct by fresh variables. The necessary variable bindings are introduced before the respective expression.[1]

EXAMPLE 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).
```

Performing the described preprocessing steps results in the following function definitions:

```
even(Arg1) ->               odd(Arg2) ->
    funcase (Arg1) of           funcase (Arg2) of
        0 -> true;                  0 -> false;
        N -> Var1 = N - 1,          N -> Var2 = N - 1,
             odd(Var1)                   even(Var2)
    end.                        end.
```

[1]This process is comparable to the introduction of variable names during A-normalization by Flanagan etal. [5]. Note that the process fixes an evaluation order for the subexpressions which is indeed implementation dependent in Erlang.

4 Flow Graphs for Erlang

4.1 Flow Graphs of Individual Functions

Flow graphs for programs using the language of Sec. 3 consist of representations of the individual expressions as nodes, and edges representing the control flow between these nodes.

There are different forms of nodes in a flow graph representing different kinds of expressions. These node forms are presented in Fig. 2.

- For computations by operators, data constructors, and data selectors, a node is generated as shown in Fig. 2.a.
- Function calls are expressed by nodes as in Fig. 2.b.
- *catch* expressions are represented as in Fig. 2.c. The circle denotes the destination for non-local return edges. The right hand side denotes the expression that might return non-locally.[2] The arrow towards the catch expression is directed towards the expression node. Both, the expression node and the circle have an outgoing edge towards the successor of the catch.
- *if*, *case*, and *funcase* expressions are represented as in Fig. 2.d. The test expressions to be matched against the patterns are stored in the `Test` fields. The `Pattern` fields hold the patterns of the individual clauses, the `Guards` fields the lists of guards, and the `ExprList` fields the graphs of the lists of expressions evaluated when the corresponding clause is chosen.

The set of nodes representing a function definition f is extended by an import node $import(a_1, \ldots, a_k)$ defining the argument variables a_i of f, a context node $context(v_1, \ldots, v_l)$ providing a local definition for all variables v_j that come from the context f was defined within, and a return node $return(R)$ expressing to leave the code of f, and to return the calculated value given in the variable *R*.

Within a single function f, directed edges are introduced from node n_1 representing an expression e_1 to node n_2 representing an expression e_2 if there is a possible control flow evaluating e_2 directly after e_1 within f.

EXAMPLE 2. *The flow graphs for the two functions* even *and* odd *defined in Ex. 1 (after preprocessing) can be extracted from Fig. 3 by deleting the dashed arrows.*

4.2 Function Calls in Flow Graphs

For function calls we have the following control flow: when reaching a node *n* representing a call to a function f, the control is transfered to the import node of f. When reaching the return node of f, the control jumps back to the (unique) node n' following *n* in the single function flow graph described above.

[2]Instead of an expression according to Fig. 2.a, other node forms, or a list of nodes are possible as right hand side as well.

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: $g \to l$
case clauses cc: p [when g] $\to l$
fun clauses fc: $(p_1,\ldots,p_k)$ [when g] $\to l$
function name fn: $\star$
expressions e: $a|X|\boxed{e_0}(\boxed{e_1},\boxed{e_2},\ldots,\boxed{e_k})|fn(\boxed{e_1},\boxed{e_2},\ldots,\boxed{e_k})|\{\boxed{e_1},\ldots,\boxed{e_k}\}|[\boxed{e_1}|\boxed{e_2}]|$
$[\boxed{e_1},\ldots,\boxed{e_k}]|$begin l end$|$if $ic_1;ic_2;\ldots ic_k$ end$|$case $\boxed{e}$ of $cc_1;cc_2;\ldots;cc_k$ end$|$
fun $fc_1;fc_2;\ldots;fc_n$ end$|$catch $e|$throw $\boxed{e}$
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. The Erlang Subset Under Consideration

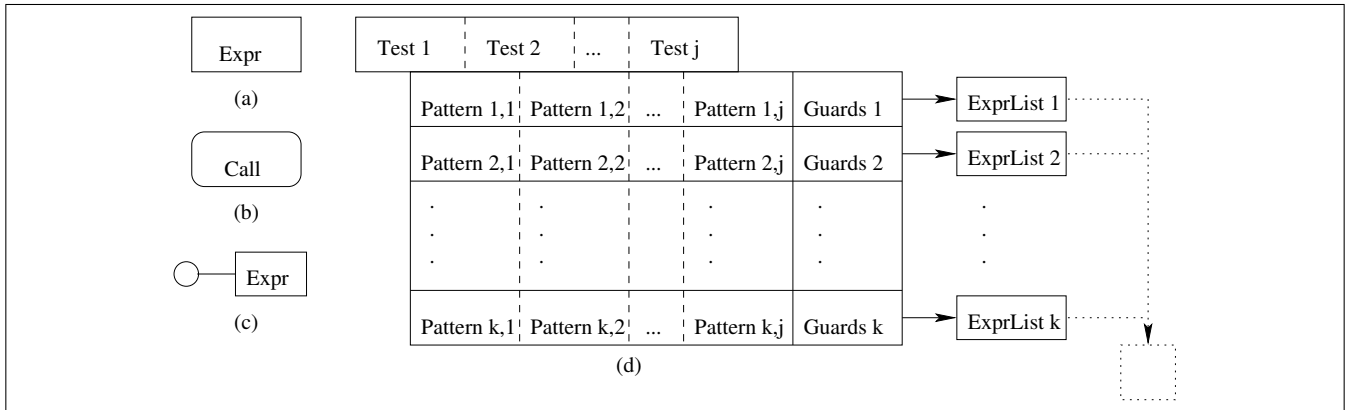

Figure 2. Flow Graph Representations of Erlang Constructs

To describe this, we introduce a new notion of *call edges* or *rubber band edges* into functional flow graphs. Such a rubber band edge (going from the calling node to the import node of the called function in the flow graph) represents the control flow to the called function and the return after reaching the return node of this function. (We can understand a function call in the flow graph as tying the arguments of a call to one end of a rubber band and throwing them to the called function while holding the other end of the band. When the computation of the call finishes, the band bounces back with the result tied to it and the computation goes on with the next node in the local flow graph.)

EXAMPLE 3. *For the even odd example given in Ex. 1 we get the graph in Fig. 3. (Rubber band edges are denoted by dashed lines.)*

4.3 Throw Edges in Flow Graphs

Besides function calls, the catch-throw-mechanism provides another way of non-local control flow in serial Erlang. When the control reaches a call to the BIF throw, it returns non-locally to the innermost catch expression in the stack of expressions currently under evaluation. After evaluating a call to throw, the computation will *never* continue with the next expression in the code.

Since the control flow from a throw node (i.e. a node representing a call to the BIF throw) to a corresponding catch node is a one way connection, there is no need to distinguish throw edges from the normal edges used in the function flow graphs.

As result, for each throw node there is *no* edge from t to its successor in the source code according to Subsec. 4.1, and for each catch node c denoting a catch expression `catch expr` such that there exists a path from the beginning of `expr` to t, not containing another catch node, an edge from t to c is inserted.

5 Data Flow Analysis Graphs

As stated by Shivers [12], the control flow given by a higher order program can depend on the data flow of the funs in the program. In this section we therefore give a definition (adapted towards the use for Erlang) of some base notions for data flow analysis that are known for imperative languages. For a definition of a variable v we write $def(v)$, for a use of v we write $use(v)$. Their precise definitions are as follows.

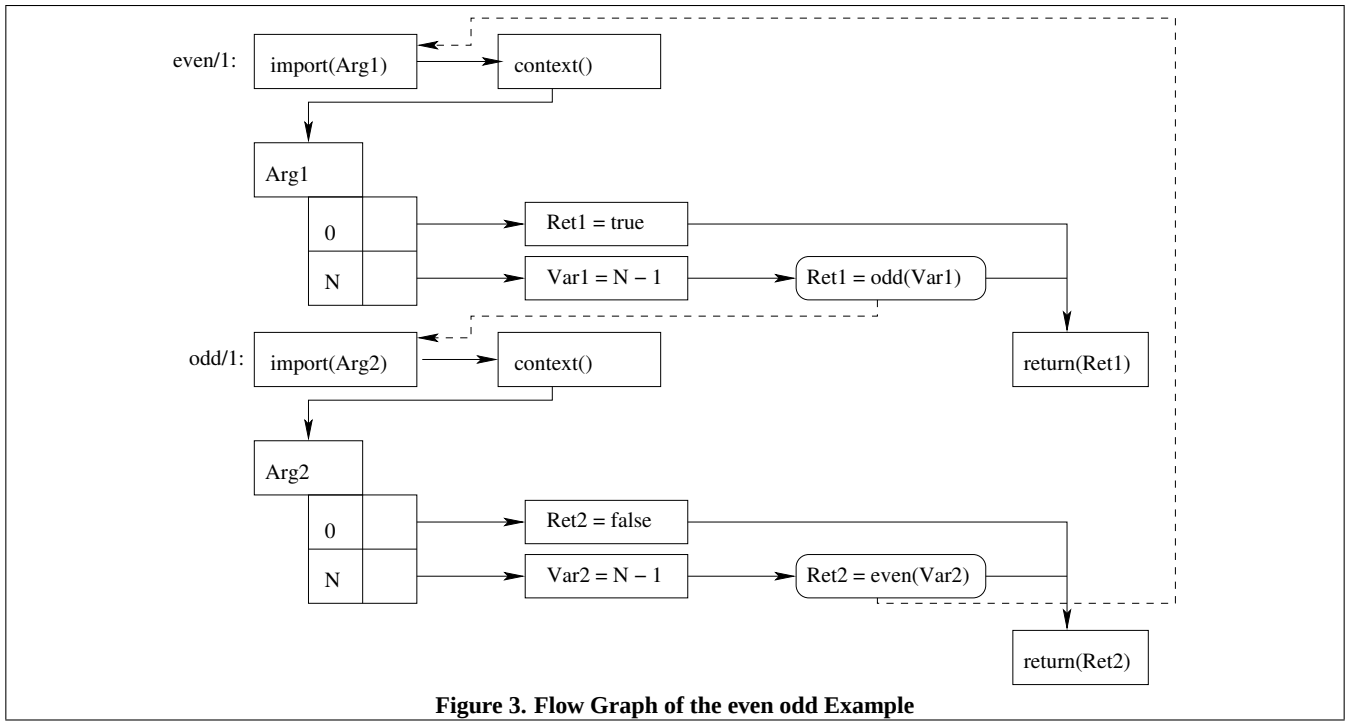

Figure 3. Flow Graph of the even odd Example

DEFINITION 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 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.*

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.

DEFINITION 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.*

DEFINITION 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 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.
- *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 v and there is a corresponding f-definition of v that reaches u.*
- *d reaches an s-use of 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 i^{th} parameter in the definition of f and a_i is the i^{th} 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. 5.

6 Flow Graph Generation

Source code oriented testing is usually done early during the software development process, and is performed to small parts of a programs source code rather than the whole program. The currently tested modules are called the *supervised modules* of a program.

Given a list of names of Erlang modules, the generation of the corresponding flow graph is done in the following steps.

1. The individual modules are read, and preprocessed according to Sec. 3.
2. The local flow graph for each function is generated.
3. The destinations of the call edges are calculated.
4. The destinations of the throw edges are calculated.

The steps (1) and (2) can be performed for each module individually. The steps (3) and (4) calculate edges that might cross module borders. They, hence, need the list of all supervised modules as input.

The most interesting step is Step (3) that calculates the possible destinations of higher order calls by data flow analysis. The process is iterated, and every step uses the call edges calculated in the steps before. As there is just a finite number of functions to be called, the iteration terminates in every case.

During Step (2) local information on the values potentially thrown by the individual functions is computed. This information is sufficient to perform Step (3) before Step (4) and to avoid the need for iteration in Step (4).

7 Test Case Analysis

For the flow graph of a sequential Erlang program as introduced above, we now want to discuss its use for evaluating test cases.[3]

7.1 Specialties of Functional Control Flow

For checking the coverage of a flow graph by a set of tests, the execution of the tests has to be performed in a supervised manner in order to collect the information about the followed execution paths.

For imperative programs an interpreter is used, that executes the tests on the flow graph, and passes every call to an unsupervised program part to the runtime system for evaluation.

In our approach unsupervised program parts can call supervised functions that were passed to them as arguments. The runtime system must, therefore, be modified to pass the control back to the interpreter when calls to supervised modules occur from outside the supervision.

7.2 Transfer of Known Coverage Criteria

In the research of testing imperative languages there exist several different coverage criteria on flow graphs. Each of these criteria defines a set of entities in the flow graph that should be covered in the best possible way by the set of performed tests.

Following Zhu etal. [15] we want to discuss some of the criteria and their use for functional programming.

- The node coverage criterion requires all *nodes* in the flow graph to be covered at least once. This corresponds to the execution of every statement in imperative programming, and to the evaluation of every expression in functional programming.
  The already existing cover tool for Erlang [4] implements a comparable criterion. It works just more coarse grain if the program source code contains several expressions in some of the lines.
- In the branch coverage criterion every *edge* in the flow graph must be covered (of course restricted to edges *within* the flow graph). When considering first order functional programs without non-local returns, we can prove that this is equivalent to the node coverage criterion. For higher order programs this criterion enforces every supervised higher order function to be called at least once in every possible supervised position. For non-local returns, every return from a *throw* expression to a *catch* expression which is a possible destination has to be performed at least once.[4]

[3] Implementing this part of the system is work in progress. Hence, broad experiences are not yet available.

[4] Because of the approximative behaviour of 0CFA, a 100 % coverage according to this criterion might be impossible. The not coverable edges can, however, be detected by the user quite easily. Especially, in the common case, a fun is generated directly as argument for a higher order function like `map`, or `fold`, no inaccuracies were expected or detected in experiments.

- Criteria based on the data flow reflect the declarative character of functional programming. We therefore expect them to be very useful in testing functional programs.[5] It is, however, a topic of future work, whether the distinction of uses in computations and in predicates carries over to the pattern matching concept of Erlang.

Liggesmeyer [7] states, that for object oriented programming, we can use the same testing tools as for structured programming, but the appropriateness of the individual tools changes and therefore should be revalidated in the new framework. A comparable revalidation is necessary for functional programming as well. For the basic control flow directed coverage criteria their use is quite obvious as they formalize minimal criteria of checking every program element at least once. Further work is possible and necessary in this area to investigate the use of criteria that are based on covering larger paths in a program (an open question here is the influence of recursion on these criteria) and criteria based on the data flow.

8 Conclusions and Future Work

By adapting the notion of flow graphs to functional programs written in a sequential subset of Erlang we have made a large step towards having the wide area of source code directed testing (which is heavily used in industry) accessible for functional programming.

Function calls have a strong influence on the control flow in functional programs comparable to the looping constructs in imperative languages. We, therefore, had to find a notion of expressing the control flow of a function call, namely the jump to a distant piece of code and the return to the calling code piece after processing the function call. The introduction of rubber band edges allows to express this situation without making the generated flow graphs unnecessarily complex.

When considering higher order functional programs, we get the additional problem that we need data flow analysis in order to determine the set of functions that is possibly called at a certain program point. We have introduced definitions and uses of a value and a notion of a definition reaching a use, that is precise enough to compute all functions that can reach a function call. Furthermore, an iterative process, essentially equivalent to 0CFA [12], allows to use the already generated flow graph for the data flow analysis task, and to update the graph correspondingly.

Future work will complete a testing tool for Erlang programs comparable to those tools already used for imperative programming. The next step in developing such a system is the implementation of a tracing tool for the test execution that copes with the problem of higher order functions escaping the control.

With the tracing completed, the adaption of different coverage criteria will become an interesting area of research: simple node or branch coverage can already be of great help in keeping the overview over testing complex and nested case distinctions. We can, however, expect further criteria that are well adapted to functional programming and that greatly increase the power of source code directed testing of functional programs.

[5] Further evidence for that statement is given by the WYSIWYT approach [9, 10, 11] using a criterion on the data flow in spreadsheets.

9 References

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

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

[3] K. Claessen and J. 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., Sept. 18–21 2000. ACM Press.

[4] *Tools version 2.3*. Documentation for Erlang OTP R9C.

[5] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

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

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

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

[9] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, 2001.

[10] G. Rothermel, L. Li, C. DuPuis, and M. 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.

[11] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239. ACM Press, June 2000.

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

[13] K. van den Berg. *Software Measurement and Functional Programming*. 1995.

[14] M. Wallace, O. Chitil, T. Brehm, and C. 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.

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