Adapting Structural Testing to Functional Programming

Manfred Widera
FernUniversität in Hagen,
58084 Hagen, Germany

Abstract—**Structural testing is heavily used in the software development process in the context of imperative programming. In order to become applicable to functional programming languages it needs, however, some adaption. We investigate the definition and generation of interprocedural flow graphs for functional programs and identify data flow oriented testing as best choice for the application to functional programming. Special data flow oriented coverage criteria are defined for the use with functional programs.**

Index Terms—**testing, flow analysis, functional programming**

I. Introduction

In order to introduce a new programming language or programming paradigm to industrial software development, it is necessary to provide tool support for the typical software engineering tasks in the new language.

Large software projects using functional languages have already been completed successfully [1]. Especially, Erlang [2] has been used in a number of very large applications. Therefore, the functional programming paradigm can be expected to get an increasing influence on industrial software. Tool support for functional programming languages offers, however, much room for improvement.

In imperative languages an important group of tools performs the validation of program fragments by structure oriented testing. This approach is based on the flow graph of a tested code fragment and is usually applied during the unit testing stage early in the software engineering process. Up to now there only exist some add-hoc approaches to structure oriented testing for functional programs, e.g. the tool cover [3] that is distributed with the Erlang programming language. This tool is, however, restricted to expression coverage without considering correspondences between distant program parts.

The work presented here aims at transferring the principles of structure oriented testing [4], [5] from the imperative to the functional programming paradigm. The main investigations necessary for this transfer are located in the area of flow graph generation and in the choice of the most appropriate coverage criteria.

The flow graph generation for functional programs is complicated by the fact that functions can be generated dynamically during the program execution, passed around in the program as ordinary data and called at distant places. Therefore, we must employ data flow analysis during the flow graph generation in an iterated manner.

In choosing the best coverage criteria for functional programs two observations are important. Functional programs [2], [6] lack most of the control flow structures control flow oriented testing in imperative languages is based on. Especially, when considering lazy functional programming languages [6], the control flow of a program is no longer easily predictable by the programmer. These two arguments motivate the investigation of data flow oriented coverage criteria, which apply to functional programs much better.

Here, we report on a project in which a structure oriented testing tool for the functional programming language Erlang [2] has been developed. Some technical details of our approach have been presented in our previous publications [7], [8], [9] and a prototype of this tool, covering the whole Erlang standard, has been implemented. The main contribution of this paper is to provide an extensive summary of the concepts used in this tool and to compare them with the structure oriented testing approaches known from imperative programming [4], [5]. Erlang was chosen from the set of modern functional languages because of its already existing relevance in industrial projects [1].

The rest of this paper is organized as follows. Section II describes the stages necessary to perform structural testing and discusses issues that need to be addressed when focusing on functional programming languages. These include specialties in generating the flow graph and a discussion why data flow oriented coverage criteria seem to be the best choice for testing functional programs. In Sec. III functional flow graphs are defined and their computation (making use of an iterated data flow analysis) is described. Section IV defines a number of data flow oriented coverage criteria and briefly discusses their use. In Sec. V related work is discussed and Sec. VI presents our conclusions.

II. Structure Oriented Testing of Functional Programs

A tool for flow graph oriented component testing consists of routines for the following subtasks.

1. The flow graph generator transforms a piece of code into a flow graph.
2. The test execution stage expects a flow graph generated by Stage (1) and a set of test inputs. For each input it computes the output and a corresponding path through the graph.
3. The coverage stage expects a flow graph from Stage (1) and a set of paths from Stage (2). It returns a set of graph constructs (depending on the actual coverage criterion) each of which is either marked as covered or as uncovered.

Each of these stages needs special adaptions towards functional programming languages and especially Erlang. The special problems occurring when considering Erlang programs for testing are described in the rest of this section.

A. Flow Graph Generation

Functional programming usually does not make use of the same control flow constructs as imperative programming languages: looping constructs are replaced by recursion, branching is not based on the results of predicates but on pattern matching. This causes the following requirements on functional flow graphs.

- Function calls must be represented in an efficient and intuitive manner.
- Flow graph constructs for presenting pattern matching are necessary. These constructs must represent the pattern matching process in an intuitive manner to the programmer/tester.

During flow graph generation, the capability of functional programming languages to express higher order functions causes additional problems.

Example 1: Consider the Erlang code fragment shown in Fig. 1. The flow graph of this fragment consists of 4 func-

```
map(F, []) -> [];
map(F, [First | Rest]) ->
  [F(First) | map(F, Rest)].

usemap(List) ->
  % expects a list of numbers and returns two
  % lists with each number incremented and
  % decremented by one, respectively.
  Inc = fun(X) -> X + 1 end, % increment func
  Dec = fun(Y) -> Y - 1 end, % decrement func
  IncList = map(Inc, List),  % incremented list
  DecList = map(Dec, List),  % decremented list
  {IncList, DecList}.        % return both lists
```

Fig. 1. Example code for usemap

tions: *map/2* (/2 denotes the arity 2) and *usemap/1* are given by their names, two further functions are generated dynamically at runtime and bound to the variables *Inc* and *Dec*. The call to F in line 2 of the function *map/2* can call both of the dynamically generated functions. Therefore, two edges must be introduced for the presentation of the possible calls.[1] □

As Shivers states [10] the control flow in a functional program can depend on the data flow during executing the program. This makes data flow analysis during the flow graph generation necessary. The iterative approach proposed by Shivers forms the basis of the flow graph generation in our system. We have adapted Shivers' approach towards the original program source code without complex program transformations. More details on the flow graph generation are described in Sec. III.

[1] Though the functions are dynamically generated at runtime their source code is known at compile time. By identifying different functions that are generated with the same source code (this is useful when the functions differ in the variable values bound in the context) we can nevertheless represent these functions in a flow graph computed at compile time.

B. Test Case Execution

The test case execution has to be performed in a supervised manner with respect to the flow graph in order to generate a path through the flow graph. Several alternatives exist to implement such a supervision.

- The compiler/runtime system of Erlang can be modified to take into account a flow graph and to provide a path together with the execution result.
- A preprocessing stage can instrument the source code to collect a path.
- An interpreter for flow graphs can be implemented, which evaluates the test calls and provides the needed paths.

For a prototypical implementation of the testing tool we have chosen the implementation of a flow graph interpreter. This approach seemed to be the easiest to implement and it offers the best extendibility towards schedule supervision for concurrent Erlang code. A detailed discussion can be found in [8].

C. Coverage Test

The most important adaption in the coverage stage is the choice of coverage criteria that are well-suited for the functional programming paradigm. In this subsection, we review the traditional coverage criteria for imperative programming and discuss their use in the context of functional programming.

C.1 Control Flow Oriented Testing

Several *structure oriented coverage criteria* for imperative programs are known in the literature [5]. Here, we discuss the use of these criteria for the testing of functional programs.

- With a given definition for functional flow graphs the adaption of the *node coverage criterion* and the *edge coverage criterion* are straightforward. Edge coverage is stronger in the context of higher order function calls[2] because a single call node can have several outgoing edges to called functions. If no higher order calls occur in a program both criteria are equivalent.
- Methods for testing loops, like the boundary interior test, are not very useful in functional programming: in the absence of looping constructs it is more important to handle function calls well in general. This is even more the case as we have no updating of variable values in functional languages. The effects of loops are, therefore, locally restricted to a higher degree because just the parameters and the result value are accessible outside a single iteration step.

From these criteria we only propose the node and edge coverage criteria for functional programming languages. One just has to note that, in contrast to imperative languages, there can be functional programs and corresponding flow

[2] I.e. the called function is given by a variable.

graphs that cannot be covered completely according to the edge coverage criterion. (This is the case because call destinations can only be computed approximately.)

C.2 Data Flow Oriented Testing

Data flow oriented testing is well-suited for the functional programming paradigm: calling a function with argument values and the returning of result values by functions describe the data flow within a program. When programming in a lazy functional language like e.g. Haskell [6] the control flow of the program is no longer easily predictable and the data flow is the only flow information the programmer can rely upon.

Analyzing the data flow oriented coverage criteria known in literature [5] yields the following situation.

- In the functional programming paradigm branching is based on pattern matching, not on predicates. The discrimination of uses into *p-uses* and *c-uses* [11] therefore does not apply.
- The *required k-tuples* criterion [12] and the *context coverage criterion* [13] check combinations of several definition-use pairs to be covered. They can be transfered to functional programming once there are appropriate definitions for checking single definition-use pairs (or du-chains as defined in Sec. IV) in functional programs.

As stated in Sec. A, data flow analysis is already necessary during the flow graph generation phase. In both, the graph generation and the coverage test, our goal is not to represent the flow of a *variable* from its definition to its use but the flow of a *value* from its generation to its use independent from the variables or structures that carry the value.

For both, the data flow analysis during the flow graph generation and the data flow oriented coverage test, we are interested in definition-use pairs that are not only based on the local definitions d, but also on the distant definitions d' defining the same value. Details on the data flow oriented coverage criteria for functional languages will be described in Sec. IV.

III. Functional Flow Graphs

A. Preprocessing of Programs

The task of presenting the control and data flow in a functional program is complicated by the presence of nested expressions in the program.

Example 2: Consider the Erlang expression `f(g(h(0)))`. In a language with eager evaluation, like Erlang, the subexpression `h(0)` is evaluated first. Its result is passed to `g` as argument and the result of this call is passed to `f`. The result of `f` is the result of the whole expression. □

In order to provide a clear description of the control and data flow by the flow graph a simple preprocessing stage (whose results can be undone for the code presentation to the programmer) enforces the *named definition property.*

Definition 1 (named definition property) An Erlang program (or program fragment) P fulfills the *named definition property* if every sub-expression of a program expression consists of a variable,[3] each function consists of a single clause with just variables as arguments and the return value of the function is bound to a return variable (which is unique for each function) on each branch of the function body. □

Example 3: Consider the following definition of the factorial function.

```
fac(0) -> 1;
fac(N) -> N * fac(N-1).
```

Preprocessing this definition to enforce the named definition property yields

```
fac(~Arg1~) ->
    case ~Arg1~ of
        0 -> ~Ret~ = 1;
        N -> ~Var1~ = N-1,
             ~Var2~ = fac(~Var1~),
             ~Ret~ = N * ~Var2~
    end.
```

□

When choosing special names for the new variables that can be distinguished from the variables occurring in the original code it is easy to undo the preprocessing transformation for presentation purposes. (E.g. in Ex. 3 each introduced variable has the form `~Name~` which cannot occur in the original program.)

B. Definition of Flow Graphs

Before describing functional flow graphs, consider the following example.

Example 4: Reconsider the preprocessed factorial function from Ex. 3. The flow graph of this function is presented in Fig. 2. In addition to the preprocessed code, the flow graph contains three additional nodes: the import node defines the formal parameters of the function, the context node provides local definitions of the variables taken from the function context (just applies to higher order functions) and the return node represents the return from the function call with the return value.

The dashed arrow denotes a call edge, i.e. it is a bidirectional edge denoting the call to a function and the return from the call. The case expression is represented by a complex node with the test expression in the first row. Each following row represents the pattern of one clause with an outgoing edge to the clause body. □

A functional flow graph $G = (V, E)$ for an Erlang code fragment C fulfilling the named definition property consists of a set V of nodes and a set E of directed edges with the following properties.

The set V of nodes is defined as follows. Each expression in C is represented by a node. Several kinds of nodes are defined to represent different expression types. (We identify the program expressions with the flow graph nodes they represent. I.e. a call node is a node representing a function call. A receive node represents a receive

[3] There is a small number of exceptions of this rule, which are difficult to describe without a precise description of the considered language. A definition showing the exceptions can be found in [7].

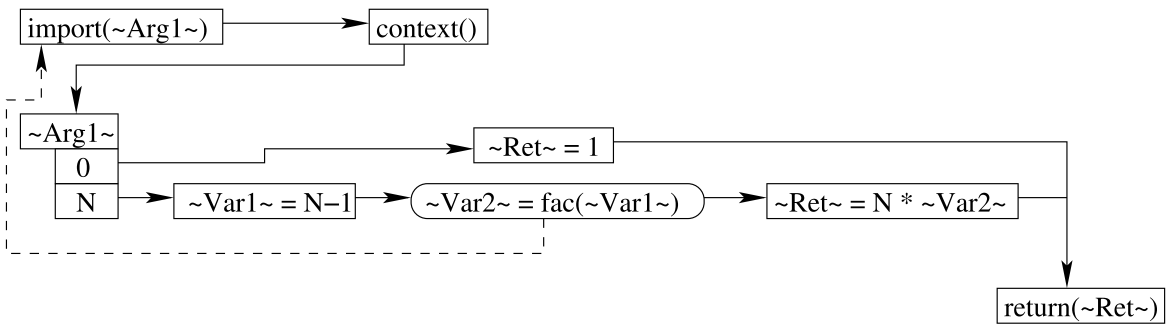

Fig. 2. Flow graph of factorial

statement, etc. E.g. Fig. 2 contains exactly one call node marked with `~Var2~ = fac(~Var1~)`.)

Additionally, there exist the following kinds of nodes for each function: an *import node* expresses a local definition for the values passed to the function as arguments during a call, a *context node* provides a local definition for the values taken from the context a (dynamically generated) function was defined within and a *return node* expresses the returning from the function call with a value bound to the return variable of the function.

Dedicated *start* and *end* nodes for the whole flow graph are not generated because there is no dedicated main function in Erlang programs. Each function that is exported from a module can serve as the entry point to the program.

Different forms of control and/or data flow are represented by different kinds of edges. The set E of edges consists of the following four edge types.

- *Neighborhood edges* represent the normal flow within a function.
- *Call edges* represent function calls. A call node has two kinds of outgoing edges (cf. recursive call to *fac* in Fig. 2).
  - Call edges lead from the call node to the possible destinations of the call. A call node with no outgoing call edge calls functions that are not part of the flow graph. In case of one or several outgoing call edges each execution of the call node at runtime chooses one of them or calls a function not represented by the flow graph.
  - A neighborhood edge from the call node to its successor in the intraprocedural flow graph represents the flow after returning from the call.

  In contrast to known definitions of interprocedural control flow graphs [14], our call edges are bidirectional and also represent the control and data flow generated by the return from the denoted call. (Having an individual edge for each possible return would make the flow graph unnecessarily complex. Furthermore, correlations between call edges and the corresponding returns would remain implicit.)
- *Throw edges* represent the non-local returns by the Erlang catch-throw mechanism. They lead from a node representing a throw to a node representing a catch. Though throw edges behave like ordinary neighborhood edges in a complete flow graph their computation needs special care.
- *Message edges* represent the data flow generated by the message passing mechanism of Erlang. They go from a send node to a receive node. These edges are special in the sense that they just present a data flow. There is no control flow between different Erlang processes.

Process generation is represented by a call edge. The spawning of a new process returns immediately with a dummy value. The new process executes independently, afterwards.

Details on the definition and the generation of flow graphs can be found in our previous work [7].

C. Generation of Flow Graphs

The generation of flow graphs for Erlang programs consists of the following steps.

1. Generation of flow graphs for the individual functions (including preprocessing of the source code).
2. Introduction of call edges, throw edges and message edges.

Stage (1) just contains some straightforward transformations of the code. In Stage (2) we employ an iterated process that loops over the following sub-tasks.

1. For each higher order function call[4] *f(v1, …, vk)* in the graph compute all definitions that reach the use of f in this call. From these definitions extract the called functions and insert a call edge from the call node to each computed destination function if the destination belongs to the flow graph.
   The call edges for first order function calls (i.e. the called function is given by its name) do not depend on any data flow and can therefore be inserted before starting the iteration.
2. For each node t describing a throw expression compute all catch nodes c such that there exists a path from c to t not containing another catch node besides c in the current graph.
3. Insert all computable message edges. Computing the message edges for a send expression s of the form $p\,!\,v$ with a process p and a value v consists of determining

[4] In contrast to other functional languages, Erlang distinguishes first order calls of functions that are defined in a module with their name and higher order calls of functions that are generated during runtime (and are usually represented by a variable).

the possible destination processes, identification of all receive statements in these destination processes and restriction of the receive statements to those whose patterns can match the sent values.

- The processes are abstracted by their initial function. For each process a set of all receive nodes it can reach is computed.
- Using data flow analysis of the variable p to the expression s identifies the possible destination processes and, hence, an initial set of possible corresponding receive statements r.
- For each possible corresponding receive statement r the patterns occurring in r are matched against the sent values. This is done by data flow analysis of the variable v to its use by the expression s and partial evaluation of the values defining v. If we cannot prove that none of the patterns of r matches any values defining v the conservative message edge generation adds a message edge from s to r to the flow graph.

The iteration over the whole sequence of sub-tasks is necessary because the control and data flow examined in one of the tasks can depend on the edges introduced during the other ones. The iteration always terminates with a fixed point, i.e. one iteration step does not change any of the edge sets.

IV. Data Flow Oriented Coverage Criteria

As already discussed in Sec. II, data flow oriented coverage criteria seem to be more appropriate for functional languages than control flow oriented criteria. Therefore, in this section we concentrate on the adaption of data flow oriented coverage to functional programs.

The base notion for all data flow oriented coverage criteria is the definition-use pair (du-pair). Its definition can be taken from the well-known approaches for imperative languages [11], [4].

Definition 2 (du-pair) Let $G = (V, E)$ be a flow graph. A *definition-use pair (du-pair)* in G is a triple (v, d, u) where v is a variable, $d \in V$ contains a definition of v, $u \in V$ contains a use of v, and there exists a path w from d to u such that v is not redefined and the scope of v defined in d is not left on w. □

Testing the coverage of du-pairs is more complicated than testing node coverage: node coverage can be tested by checking every program expression for coverage without computing a flow graph. Therefore, we are interested in the additional strength in testing provided by the du-pairs. To measure this we define non-trivial du-pairs as follows.

Definition 3: Let G be a flow graph and P the set of du-pairs in G. A du-pair $(d, u) \in P$[5] is non-trivial if there exist du-pairs $(d, u'), (d', u) \in P$ with $d \neq d'$ and $u \neq u'$. □

Informally, a du-pair is non-trivial if it is neither determined completely by its source node nor by its destination node. Such a non-trivial du-pair p can be overlooked by a test set T even though T fulfills the node coverage criterion for the graph containing p. Therefore, a large number of non-trivial du-pairs implies the need for checking some data flow oriented coverage criterion.

In measurements with our prototype [9], between 25 % and 40 % of all du-pairs were non-trivial and carried information not covered by the node coverage criterion. This additional strength makes the effort of computing a flow graph for testing valuable.

The data flow criteria are not comparable to the edge covering criterion: a call edge whose destination function does not expect an argument does not represent any data flow (i.e. the data flow criteria do not subsume edge coverage). Vice versa, single edges are not able to represent every form of data flow (i.e. edge coverage does not subsume data flow oriented coverage). Since both, edge coverage and the data flow criteria need the computation of the flow graph, a chosen data flow criterion should be combined with the edge coverage criterion for maximum testing strength.

Definition-use pairs of the form defined above just describe the flow of a data object as long as it is bound to a certain variable. By considering sequences of du-pairs (called *du-chains*) we are able to express several additional ways of data flow that are also of high interest in functional programs.

- A value can be passed from one variable to another. This is possible explicitely by a copy expression $A = B$ or implicitly for the arguments of a function call and the parameters of the called function. To express this, we define *aliasing aware* du-chains in Def. 4.
- Functional programming languages provide standard data constructors for constructing e.g. lists or tuples. A value from a variable v can be stored in a structure at a node n_1 and selected from the structure at a distant node n_2 defining a variable v'. With a data flow of the structure from n_1 to n_2 given, we want to represent the data flow from the definition of v to the use of v'. This will be expressed by *structure aware* du-chains.
- Many values are generated within calls to subfunctions. We are interested in the flow of the data objects within the called function to the return from the call and further to the use within the calling function. To express this, we define *result aware* du-chains.
- Functions that are constructed dynamically during runtime can make use of the variables that are accessible when the function is constructed. The values of these variables are stored in a lambda closure. When such a lambda closure f is generated in a node n_1 the value of a variable v is frozen. It is available within the function after f has been called at a distant node n_2. A local definition for v is provided by the context node of f. With the data flow of f from n_1 to n_2 given, we are interested in the data flow of v from the definition outside of f to the use inside of f. We express this by *freeze aware* du-chains.

Formally, a du-chain is a sequence of du-pairs such that its

[5] We often omit the variable v of a du-pair if it is not of interest.

first definition and its last use denote the same value.

Definition 4 (du-chains) Let G be a flow graph. A du-chain is a sequence (with sequencing operator ;) of du-pairs defined as follows.

- Each du-pair in G is a du-chain in G.
- Let $e : v' = v$ be a copy expression in G with variables v and v'. Let d_1 be a du-chain in G ending with the use of v in e and d_2 a du-chain in G starting with the definition of v' in e. Then $d_1; d_2$ is an *aliasing aware* du-chain in G.
- Let $e_1 : s = \{v_1, \ldots, v_k\}$ be a tuple construction and $e_2 : v' = element(i, s')$ a tuple selection. Let d_1 be a du-chain in G ending with the use of v_i in e_1, d_2 a du-chain in G starting with the definition of s in e_1 and ending with the use of s' in e_2 and d_3 a du-chain in G starting with the definition of v' in e_2. Then $d_1; d_2; d_3$ is a *structure aware* du-chain in G.[6]
- Let e_1 be a function call and e_2 the return node of a function f reached by a call edge from e_1. Let d_1 be a du-chain in G ending with the use of the return variable v of f in e_2 and let d_2 be a du-chain in G beginning with the binding of a variable v' to the call result of e_1. Then $d_1; d_2$ is a *result aware* du-chain in G.
- Let e_1 be the generation of a lambda closure f in G. Let $e_2 : f'(a_1, \ldots, a_k)$ be a higher order function call in G, and e_3 the context node of f defined in e_1. Let d_1 be a du-chain in G ending with the use of a variable v in e_1,[7] d_2 a du-chain in G starting with the definition of f in e_1 and ending with the use of f' in e_2 and d_3 a du-chain in G starting with the definition of v in e_3. Then $d_1; d_2; d_3$ is a *freeze aware* du-chain in G.

In all cases, pattern matching in branching constructs is handled in analogy to the matching operator =. □

The presented awareness levels of du-chains can be combined with each other: the awareness level of the du-chains one wants to define determines the cases of Def. 4 to use and restricts the sub-chains in these cases to the same awareness level. Defining e.g. the set of all aliasing and result aware du-chains we choose the cases for aliasing aware and for result aware du-chains and in both cases d_1 and d_2 can themselves be aliasing and result aware du-chains.

Aliasing awareness and result awareness reflect the importance of function calls in functional programming: in Fig. 2 the call edge implicitly expresses a copy expression `~Arg1~ = ~Var1~` between the argument of the call and the parameter of the called function. This can be expressed by an aliasing aware du-chain. After the recursive call finishes the result (bound to `~Ret~`) is assigned to `~Var2~`. This data flow can represented by a result aware du-chain.

Especially the aliasing awareness is, furthermore, important to analyze loops. Loops are expressed by recursive function calls, and the termination of the recursion usually depends on one or several of the functions arguments. Aliasing awareness is the only way to express the flow of data from the computation of the argument for the next recursion step (`~Var1~ = N - 1` in Fig. 2) to its use for the termination control (use of `~Arg1~` in the example).

The remaining levels of structure awareness and freeze awareness are based on the special properties and constructs occurring in functional languages and have the goal to deduce the *values* reaching a use as precisely as possible.

Our measurements for sequential Erlang modules [9] showed that a combination of aliasing, structure and freeze awareness is feasible in every case. Compared to checking individual du-pairs only, it is not much more complicated to check du-chains of this level. When adding result awareness, the number of du-chains can become infeasible (especially for recursive functions which contain many different clauses and which return structured values).

We came to the conclusion that testing the coverage of all aliasing, structure and freeze aware du-chains is valuable in every case and to add result awareness whenever this yields a feasible number of du-chains.

V. Related Work

The concept of generating flow graphs for higher order programs is described by Shivers [10] proposing several different levels of precision for the needed data flow analysis. These different precision levels are further analyzed by Ashley/Dybvig [15]. Especially, the level 0CFA is similar to our approach. Most of the CFA approaches, however, do not focus on the presentation of the generated flow graphs to the programmer.

Some approaches on testing and debugging functional programs have been proposed. QuickCheck [16] aims at automatically testing Haskell programs by generating input data on a random basis and checking the results with constraints on the expected output.

In the WYSIWYT framework [17], [18], [19] flow graphs are used for testing spreadsheets, which are considered as first order functional programs without recursion.

The module cover that comes with the tools library of Erlang [3] implements a coverage test for Erlang source code that analyzes the individual lines of the source code for coverage. Cover does not need to employ data flow analysis since it directly works on the program source code. As a drawback, it is 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.

There are several works presenting data flow analysis in an imperative framework (e.g. [20]). The presentation given there is more general than needed by our approach and describes several forms of data flow analysis. The tool used by us is called *reaching definitions analysis* there.

Several works on data flow oriented testing for imperative programming include those already discussed in Sec. II ([13], [12], [11]). The problem of infeasible paths in a flow graph occurring in these (and our) approaches is addressed by introducing new criteria just considering feasible paths

[6] Analogously, structure aware du-chains can consider pairs instead of tuples and selections by pattern matching instead of predefined selection functions.

[7] A lambda closure generation is a use of each variable reaching it.

[21]. This, however, makes the question whether a test set fulfills one of these criteria undecidable.

The combination of flow graphs of individual procedures/functions to an interprocedural control flow graph (ICFG) was developed by Landi and Ryder [14]. A call corresponds to two nodes, one for the call and one for the return from the call, respectively. Edges lead from the call node to the start node of the called procedure and from its end node to the return node in the calling procedure. This leads to the problem of non-realizable paths, i.e. paths whose call edge and return edge do not correspond to the same call site.

Problems related to the analysis of higher order programs also arise when considering function pointers, e.g. during a points-to analysis [22]. The approach of Emami Ghiya and Hendren is comparable to our approach in using an incomplete data structure for iterating the necessary analysis but it does not cover exception handling and message passing as they occur in Erlang.

Program analysis in the presence of exception-handling is discussed by Sinha and Harrold [23] for Java, i.e. not in the context of higher order programming. Their approach can be incorporated into our Erlang flow graphs for covering the most recent changes in the Erlang standard [24].

VI. Conclusion

We have shown that structure oriented coverage is applicable to functional programming languages. A number of adaptions is, however, necessary. In defining flow graphs the importance of function calls has to be reflected by a simple representation. We have introduced call edges such that on reaching a function call the control follows a call edge first; when reaching the return node of the called function it bounces back along the call edge and goes on by following the outgoing neighborhood edge of the call.

Computing flow graphs for functional programs is more complicated than it is the case for imperative programs: since the possible control flow in a program depends on the data flow in higher order programs, data flow analysis is necessary for computing call edges. An iterated process performs this data flow analysis.

Analyzing different coverage criteria known for imperative languages, we have identified data flow oriented criteria as the best choice for functional programs. Instead of distinguishing p-uses and c-uses as known from literature [11] we have introduced the notion of du-chains, i.e. sequences of du-pairs with their first definition and their last use referencing the same value. Definition use chains, hence, represent the flow of a value through different variable and structure bindings.

Different levels of du-chains have been defined and measurements comparing the individual levels have led to a simple standard procedure choosing the right level for testing a given program fragment.

References

[1] Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G.: AXD 301: A new generation ATM switching system. Computer Networks (Amsterdam, Netherlands: 1999) **31** (1999) 559–582

[2] Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in ERLANG. 2nd edn. Prentice Hall (1996)

[3] NN: Tools version 2.3. (2003) Documentation of Erlang/OTP R9C.

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

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

[6] Jones, S.P., ed.: Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press (2003)

[7] Widera, M.: Flow graphs for testing sequential Erlang programs. In: Proceedings of the 3rd ACM SIGPLAN Erlang Workshop, ACM Press (2004)

[8] Widera, M.: Flow graph interpretation for source code directed testing of functional programs. In Grelck, C., Huch, F., eds.: 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)

[9] Widera, M.: Data flow coverage for testing Erlang programs. In van Eekelen, M., ed.: Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP'05). (2005)

[10] Shivers, O.: Control-flow analysis in Scheme. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. (1988) 164–174

[11] Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering **11** (1985) 367–375

[12] Ntafos, S.C.: On required element testing. IEEE Transactions on Software Engineering **10** (1984) 795–803

[13] Laski, J.W., Korel, B.: A data flow oriented program testing strategy. IEEE Transactions on Software Engineering **9** (1983) 347–354

[14] Landi, W., Ryder, B.G.: Pointer-induced aliasing: a problem classification. In: POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1991) 93–103

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

[16] Claessen, K., Hughes, J.: 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., N.Y., ACM Press (2000) 268–279

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

[18] Rothermel, G., Li, L., DuPuis, C., Burnett, M.: 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, IEEE Computer Society Press/ACM Press (1998) 198–207

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

[20] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (1999)

[21] Frankl, P.G., Weyuker, E.J.: An applicable family of dataflow testing criteria. IEEE Transactions on Software Engineering **14** (1988) 1483–1498

[22] Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, New York, NY, USA, ACM Press (1994) 242–256

[23] Sinha, S., Harrold, M.J.: Analysis of programs with exception-handling constructs. In: Proceedings of the International Conference on Software Maintenance. (1998) 348–357

[24] Ericsson Utvecklings AB: Erlang Reference Manual, Version 5.4.9. (2005)