Data Flow Coverage for Testing Erlang Programs

Manfred Widera

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

Abstract

Flow graph oriented testing is heavily used in industry, but has not yet been adapted to functional programming. Carrying over this approach from imperative programs to other programming paradigms consists of adapting the notion of flow graphs, and the identification of useful coverage criteria. The identification of coverage criteria is the topic of this paper. We define a number of data flow oriented coverage criteria, show that they carry more information than the node coverage criterion, which is the simplest criterion known for imperative programming, and compare the different data flow oriented coverage criteria with each other.

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. Compared to the test case selection based on the intended I/O-behavior of a program, these structure oriented criteria have the closest correspondence to the *actual implementation* under testing. Structure oriented testing is usually applied to small program fractions like single modules, and is an important part of the early stages of software development.

In the context of functional programming languages, there are just some ad hoc approaches on source code directed testing, as e.g. the tool cover [ET] that checks the individual *lines* of a program in the functional language Erlang for coverage. This tool works by an instrumentation of the the source code of a module that is focused on the executable lines; it is therefore not extendible to take into account relationships between distant parts of the program, e.g. the data flow, and even more the concurrent data flow in an Erlang program.

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. It is important to note, that systematic testing cannot be replaced completely by employing the suitability of functional languages for verification. As the first main reason, verification is a quite expensive, and time consuming task, and cannot be applied to all the less critical

components (which should nevertheless be as correct as possible). Second, verification is always done against an already formalized specification of the intended program behavior which itself is not guaranteed to be correct.

Based on the already available notion of flow graphs for functional programs [Wid04] the aim of this work is to present a range of data flow oriented coverage criteria for functional programming. Different data flow oriented coverage criteria are defined which are based on the ideas of data flow based testing for imperative languages [ZHM97], but need some adaption towards the functional programming paradigm. These criteria are compared with the node coverage criterion, which forms the most basic control flow oriented coverage criterion known for imperative programming [ZHM97].

The rest of this work is organized as follows. In Sec. 2 we discuss some approaches related to the current work. Section 3 contains the preliminaries the current work is based upon. In Sec. 4 some notions of data flow analysis are recalled. They are used in Subsec. 5.1 for defining data flow oriented coverage criteria. Subsection 5.2 sketches some measurements of different coverage criteria, and discusses their use. In Sec. 6 we draw the conclusions, and discuss related work.

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 concept 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 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 [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. 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.

There are several works presenting data flow analysis in an imperative framework (e.g. [NNH99]). The presentation given there is more general than needed by our approach, and describes several forms of data flow analysis. The tool used in our approach is called *reaching definitions analysis* there.

3 PRELIMINARIES

The language under consideration essentially consists of the whole Erlang standard. For the presentation in this work, we, however, concentrate on the subset defined in Fig. 1 (ignoring the boxes around some expressions for the moment). Definitions consisting of a $\star$ are not of interest here, and are therefore omitted. Infix operators are considered as ordinary functions. Timeouts for receive expressions are omitted for simplification reasons. The BIFs *throw/1* and the binary operator *!* (which is denoted as ordinary function *send/2*) need a special treatment and are therefore considered as syntactic keywords in this work.

Some concepts are special to Erlang, or their notions differ from the usual ones: 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 $e_0(e_1, e_2, \ldots, e_k)$ with an expression e_0. The lambda closures occurring for e_0 are called *funs* in the context of Erlang due to the use of the keyword `fun` for their generation.

In the rest of this paper programs are assumed to fulfill the following *named definition property* which is easy to obtain by a preprocessing stage.

Definition 1 (named definition property). *A program P fulfills the* named definition property *if*

- *Every sub-expression of a program expression that is boxed in Fig. 1 consists of an instantiated variable.*
- *Each function consists of a single clause with just variables as arguments.*[1]

[1] Special care is necessary for the case distinction given by the different function

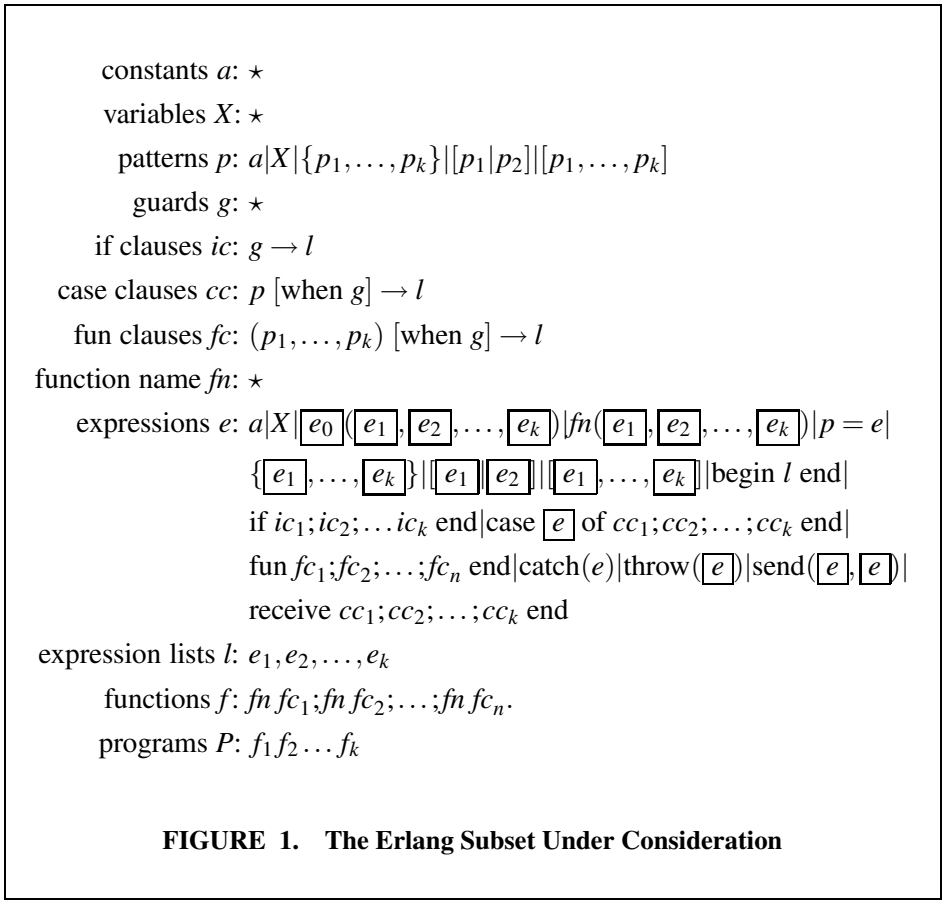

FIGURE 1. The Erlang Subset Under Consideration

- *The return value of the function is bound to a return variable on each branch of the function body. The return variable is unique for each function.*

The preprocessing stage enforcing the named definition property yields a name for each use of a value, a property that is helpful for performing data flow analysis, and for presenting the data flow results. When choosing special variable names, that can be distinguished from the variables originally occurring in the program, it is easy to undo the preprocessing changes for presenting the program or the generated flow graph to the programmer.

For a program P fulfilling the named definition property, a flow graph $G = (V,E)$ consists of the a set V of nodes and a set E of edges determined by P as follows.

- Each expression in the program is represented by a node. Several kinds of nodes are defined to represent different expression types. In addition to the

clauses. They can be expressed by a *case* expression inside the single clause. For functions with arity > 1 the arguments and the corresponding patterns must, however, be structured into tuples of equal element number.

nodes representing expressions, there are the following kinds of nodes for each function.

  - an import node provides local definitions for the values passed to the function as arguments during a call.
  - a context node provides local definitions for the values taken from the context, a fun was defined within. For the functions defined on top level in a module the context node is empty.
  - a return node expresses the returning from the function call with a value bound to the return variable of the function.

- Neighborhood edges represent the normal control and data flow within a function.
- Call edges represent function calls. They lead from a call node (a node representing a function call) to the import node of the called function. There is no additional edge for the return from the call. The control and data flow generated by the return is represented by the call edge in addition to the call itself.
- Throw edges represent the non-local returns by the Erlang catch-throw mechanism. Throw edges go from a node representing a throw to a node representing a catch.
- Message edges represent the data flow generated by the message passing mechanism of Erlang. They go from a node representing a send to a node representing a receive.

The detailed definition and generation of flow graphs is omitted here. The basic concepts can be found in the literature [Wid04].

In the rest of this paper we will identify flow graph nodes, and the program expressions they represent.

4 DATA FLOW ANALYSIS

As stated by Shivers [Shi88], the control flow given by a higher order program can depend on the data flow of the funs from their generation to their application in the program. In this section we, therefore, summarize the definitions (adapted towards the use for functional flow graphs) of some base notions of data flow analysis, that are known for imperative languages [NNH99]. For a definition of a variable v we write $def(v)$, for a use of v we write $use(v)$. Most of the definitions of these notions have already been presented in the literature [Wid04].

We just want to recall some special forms of uses and definitions, that form corresponding pairs.

- Corresponding f-uses and f-definitions express the situation of using a definition for freezing it in a fun generation. It is defrosted by the corresponding definition in the context node of the function.[2]

- Corresponding s-uses and s-definitions express the situation of using a value to store it in a structure, and selecting it from there in the definition of a variable.

- Corresponding m-uses and m-definitions express the use of a value for sending it as a message, and the definition of a variable by receiving this message.

Each of these definitions corresponds to several uses in general, and vice versa. For an s-use, and the corresponding s-definition, or for an m-use, and the corresponding m-definition the variable names usually differ.

Local data flow information is stored in form of definition use pairs (du-pairs).

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, $d \in V$ is a definition of v, $u \in V$ is a use of v, and there exists a path w from d to u such that v is not redefined on w.*

When presenting du-pairs we usually omit the variable v, but only denote the definition node and the use node. The considered variable can normally be extracted from the context when necessary.

5 DATA FLOW COVERAGE

The presentation of the data flow oriented coverage criteria for test sets is divided into two main parts. In Subsec. 5.1 the main definitions of this work present the data flow oriented coverage criteria for test sets. These coverage criteria are compared with each other and with simple control flow oriented coverage criteria in Subsec. 5.2.

The two parts differ in the covered Erlang subset. While the definition of the coverage criteria is done for the whole Erlang standard, the comparison is based on example modules, that do not make use of the concurrent language constructs. The comparison, hence, just provides results for coverage criteria that make sense in the context of sequential functional programming.

5.1 Definition of the Coverage Criteria

We define a number of different data flow oriented coverage criteria. The different criteria are based on the coverage of du-chains, i.e. sequences $p_1;\ldots;p_k$ of du-pairs, such that the definition of p_1 and the use of p_k denote the same value. Describing distant definitions and uses of *values* instead of sticking to the *variables*

[2] It is important to note here, that a fun generation represents a use of each definition, that is used within the body of the fun. For the local uses, the external definition is hidden by a corresponding definition in the context node.

they are bound to, is one of the main adaptions of the data flow criteria known from imperative languages towards functional programming. The individual criteria differ in the form of the du-chains under consideration.

Definition 3 (du-chains). *Let G be a flow graph. A du-chain is a sequence (with sequencing operator ;) of du-pairs. The set of all* du-chains *in G is defined as follows.*

- *Each du-pair in G is a du-chain in G.*
- *Let* $e : \mathtt{v}' = \mathtt{v}$ *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 (a-aware) *du-chain in G.*
- *Let* $e_1 : \mathtt{s} = \{\mathtt{v_1}, \ldots \mathtt{v_k}\}$ *be a tuple construction containing an s-use of a variable* v_i*, and* $e_2 : \mathtt{v}' = \mathtt{element}(\mathtt{i}, \mathtt{s}')$ *a corresponding s-definition of a variable* v' *by 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 (s-aware) *du-chain in G.*[3]
- *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 (r-aware) *du-chain in G.*
- *Let* e_1 *be the generation of a fun f in G containing an f-use of a variable v. Let* $e_2 : \mathtt{f}'(\mathtt{a_1}, \ldots, \mathtt{a_k})$ *be a higher order function call in G, and* e_3 *the context node of the fun f defined in* e_1*, i.e.* e_3 *contains a corresponding f-definition of v. Let* d_1 *be a du-chain in G ending with the use of a variable v in* e_1*,* 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 (f-aware) *du-chain in G.*
- *Let* $e_1 : \mathtt{send}(\mathtt{v_1}, \mathtt{v_2})$ *be a send expression in G containing an m-use of the variable* v_2*, and* e_2 *a receive statement in G containing a corresponding m-definition of a variable v, i.e. there exists a message edge from* e_1 *to* e_2 *in G. Let* d_1 *be a du-chain in G ending with the use of* v_2 *in* e_1*, and* d_2 *a du-chain in G starting with the definition of v in* e_2*. Then* $d_1;d_2$ *is a* message aware (m-aware) *du-chain in G.*

[3] s-aware du-chains can also consider pairs instead of tuples, and selections by pattern matching instead of predefined selection functions. These cases are defined in an analogous manner.

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

The definition above provides a number of different awareness levels of du-chains that can be combined with each other. The properties of the sub-chains needed to define a certain du-chain are left open in Def. 3. The awareness level of the du-chains one wants to define determines the cases of Def. 3 to use, and restricts the sub-chains in these cases to the same awareness level. Defining e.g. the set of all aliasing aware, and message aware du-chains, we choose the cases for aliasing aware, and for message aware du-chains, and in both cases d_1, and d_2 can itself be aliasing aware, and message aware du-chains.

For the short presentation of a combination of awareness levels we just concatenate the letters, taken from the set $\{a,s,r,f,m\}$ for their presentation. E.g. du-chains that are aliasing aware and message aware are called *am-aware*.

Definition 4 (traces in flow graphs). *Let P be a program, $G = (V,E)$ the flow graph of P, and c a test case. The* trace t generated by evaluating c in G *is a sequence of nodes in V that represents the control flow in G during evaluating t. For each pair of two subsequent nodes (v_1,v_2) in t one of the following cases is fulfilled.*

1. *There is a neighborhood edge, a call edge, or a throw edge from v_1 to v_2 in E.*
2. *v_1 is a return node of a function f, v' is the import node of f, v is a call node such that there is a call edge from v to v', and there is a neighborhood edge from v to v_2.*[4]

A part of a trace, that begins with the destination of a call edge, and ends with the corresponding return node v_1 according to case (2) above, is put in parentheses.

Using this definition of a trace we can now define the notion of a test case covering a du-pair or du-chain. Informally, a test case covers a du-pair, if the trace corresponding to the test case contains a sub-path from the definition to the use of the pair without a valid redefinition of the value.

Definition 5 (coverage of du-pairs). *Let G be a flow graph, p a du-pair in G and t a test case. t covers p if the trace generated by evaluating t contains a path in G from the definition of p to the use of p. The only redefinition of the value considered in p may occur inside of parentheses.*

A du-chain consists of a list of du-pairs. To cover such a du-chain, all the pairs need to be covered in the right order, and without an interruption.

Definition 6 (coverage of du-chains). *Let G be a flow graph, $c = p_1;p_2;\ldots;p_k$ a du-chain in G and t a test case. t covers c if the trace generated by evaluating t contains a subtrace $t_1;n_1;t_2;n_2;\ldots;n_{k-1};t_k$ such that*

[4] This situation represents the return from a call to f in v.

- *all t_i are traces in G such that the final node of t_{i-1} and the first node of the subsequent t_i differ,*
- *the n_i are either nodes or empty traces ε, and*
- *one of the traces $n_{i-1};t_i;n_i$, $t_i;n_i$, or $n_{i-1};t_i$ covers the du-pair p_i.*

In case of traces t' with brackets, the last node before an opening bracket (i.e. the call node) can be repeated after the corresponding closing bracket to form the trace t.

The n_i in the definition above can play two roles:

- When, for instance, considering an a-aware du-chain $p_1;p_2$ with an expression $A = B$ representing the last node of p_1 and the first node of p_2, the node $A = B$ is represented by n_1 and can, therefore, be used in the coverage of both p_i.
- Considering an f-aware du-chain $p_1;p_2;p_3$, the use in p_2 is given by the call of a fun. The definition in p_3 represents a context node. Concatenating a trace t_2 covering p_2 and the trace t_3 covering p_3 does not yield a valid trace in G, because after each function call, the import node n of the called function is reached before the context node. The definition above allows to consider the trace $t_2;n;t_3$ instead.

Example 7. 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. 2. Call edges are denoted by dashed lines. The numbers at the right top corners of the nodes are the unique node numbers.

In this flow graph, the sequence $d_1 := (1,3);(3,5)$ is an a-aware du-chain; $d_2 := (13,14);(6,7)$ and $d_3 := (11,14);(6,7)$ are r-aware du-chains.

The trace for the call `even(1)` is $(1,2,3,5,6,(8,9,10,11,14),7)$. This trace covers d_1 and d_3, but not d_2.

Definition 8 (data flow coverage criterion). *Let G be a flow graph and T a set of test cases. T fulfills the xy-aware[5] data flow criterion for G, if for each xy-aware du-chain c, there exists a $t \in T$ such that t covers c.*

Compared to testing one of these data flow based coverage criteria, it is much easier to test the control flow based node coverage criterion known for imperative programming languages [ZHM97]. In functional languages it suffices to check, whether each expression in the program is executed at least once. This can be

[5] $xy \subseteq \{a,s,r,f,m\}$ denotes an arbitrary awareness level.

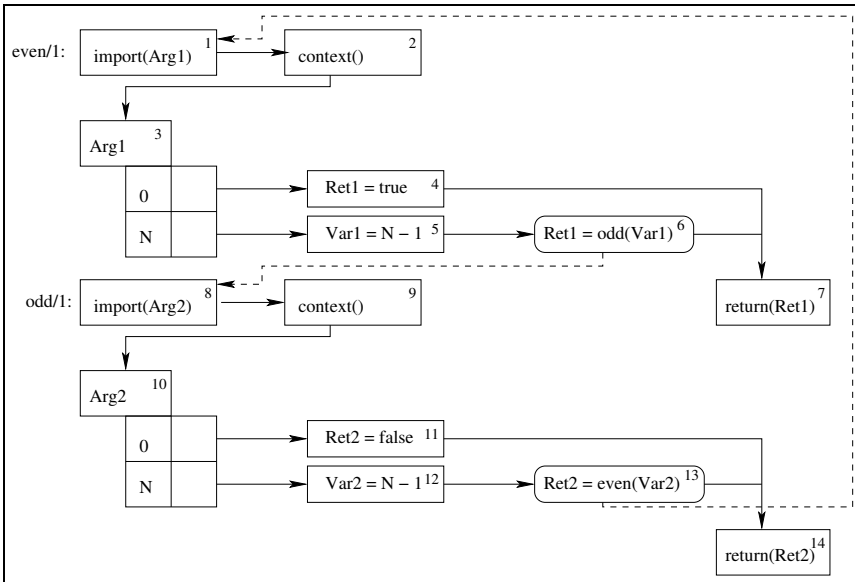

FIGURE 2. Flow graph of the even-odd example

done even without computing a flow graph. By a comparison we want to show that the data flow oriented criteria carry significantly more information and therefore deserve the additional effort. The following definition recalls the node coverage criterion.

Definition 9 (node coverage criterion). *Let $G = (V, E)$ be a flow graph, and T a set of test cases. T fulfills the node coverage criterion for G, if for each $v \in V$ there is a $t \in T$ such that the trace generated by evaluating t contains v.*

5.2 Using the Coverage Criteria

For determining the use of the individual data flow oriented coverage criteria, we want to compare them with each other and with the node coverage criterion. In Subsec. 5.2.1 we present the properties of some test modules that are used to compare the different criteria. Subsection 5.2.2 contains the comparison of the data flow criteria with the node coverage criterion, and in Subsec. 5.2.3 the data flow oriented coverage criteria are compared with each other.

5.2.1 Presentation of the Test Modules

For comparing the different coverage criteria, we have chosen the following three modules.

- *numbering_t.erl* is derived from the module *numbering.erl*. It is part of the flow graph generation system for Erlang programs. It is called after reading

and preprocessing a module. It traverses the parse result of a given module, introduces node numbers for the nodes, collects a number of local data flow information, and maintains some additional data structures for quick access to certain parts of the flow graph.

- *compute_throw_edges_1.erl* is derived from *compute_throw_edges.erl*, which computes and inserts the throw edges (i.e. edges from a throw node to a catch node) during computing the flow graph of an Erlang program.

- *gb_trees* is a module providing balanced binary trees as data structure. It is part of the standard library of OTP-R9C.

numbering.erl, and *compute_throw_edges.erl* were infeasible when generating the asr-aware and asfr-aware du-chains. Therefore, we simplified them by eliminating some non-recursive cases from large functions. This yielded *numbering_1.erl*, and *compute_throw_edges_1.erl*, respectively.

Some statistical information on these modules is presented in Table 1. The count for the lines of code just contains semantics carrying lines of code, no comment lines, and no whitespace lines.

A du-chain is computed starting with its final du-pair and extending it by further du-pairs fitting according to the chosen awareness level. Usually, there are several du-pairs fitting in a certain situation. Therefore, we usually get more du-chains when switching to a higher awareness level.

Example 10. Consider a copy expression $e : B = A$, and a du-chain c starting with the definition of B in e. Each du-pair representing a definition reaching the use of A in e can extend c.

However, a higher awareness level generates a number of redundant du-chains in the following sense: a du-chain c' is redundant, if there exists a further du-chain c such that c' forms a prefix of c. This is always true for an du-chain c' whose last du-pair p was added to another du-chain c. In Tab. 1 and in the remaining measurements, redundant du-chains are not considered. This can cause the number of du-chains to *decrease* when switching to a higher awareness level.

5.2.2 Node Coverage versus Data Flow Coverage

In principle, we have the following correspondence between the data flow oriented criteria and the node coverage criterion: as long as every defined value is also used in a flow graph, each of the data flow oriented coverage criteria subsumes the node coverage criterion. This is the case, because each node occurs in a du-pair, and therefore in each set of du-chains.[6]

[6]Just considering flow graphs with every definition being used is not a real restriction, because we are not interested in unnecessary code. The du-pair generation stage can easily raise a warning for each unused definition.

	P_1	P_2	P_3
number of lines	1187	324	224
number of nodes	2959	526	690
$\emptyset$-aware chains	2804	320	391
a-aware chains	1787	218	379
af-aware chains	1783	302	379
as-aware chains	1755	318	379
ar-aware chains	1829	313	453
afs-aware chains	1751	302	379
afr-aware chains	1826	297	453
asr-aware chains	5298	28037	467
afsr-aware chains	7506	27979	467

P_1: numbering_t, P_2: compute_throw_edges_t, P_3: gb_trees

TABLE 1. The test modules in numbers

	du-pairs	non-trivial du-pairs	ratio
numbering_t	1804	686	38,0 %
compute_throw_edges_t	320	95	29,7 %
gb_trees	391	96	24,6 %

TABLE 2. Relation between nodes and du-pairs

The question remains, whether data flow oriented coverage is more useful than the node coverage criterion. To measure this we define non-trivial du-pairs as follows.

Definition 11. *Let G be a flow graph, and P the set of du-pairs. A du-pair* $(s,d) \in P$ *is non-trivial if there exist du-pairs* $(s,d'),(s',d) \in P$ *with* $s \neq s'$ *and* $d \neq d'$.

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.

The measurements of non-trivial du-pairs for our three example modules is given in Table 2. These measurements show, that about 25 % – 40 % of all du-pairs are non-trivial and carry information not covered by the node coverage criterion. Checking the data flow oriented coverage criteria is more complicated, especially because it makes the computations of a flow graph for the tested modules necessary. We, nevertheless, believe that especially the ratios of non-trivial du-pairs (which

are the *simplest* form of du-chains) in complex modules make the additional effort of checking a data flow oriented coverage criterion valuable.

5.2.3 Comparing the Different Data Flow Criteria

For discussing the usefulness of the individual data flow oriented coverage criteria we will present a number of measurements for the modules described in Subsec. 5.2.1, and the for different awareness levels in this subsection.

The intention to measure the additional testing strength of du-chains with higher awareness levels leads to the notion of *trivial du-chains*.

Definition 12 (trivial du-chains). *Let G be a flow graph, C_1 the set of all xy-aware du-chains of G, and C_2 the set of all xyz-aware du-chains of G (*$\mathrm{xy} \subset \mathrm{xyz} \subseteq \{a,s,r,f,m\}$*). A du-chain $d_2 \in C_2$ is trivial with respect to C_1, if there exists a $d_1 \in C_1$ such that*

- *d_2 contains d_1 as a subsequence, and*
- *No other du-chain $d_2' \in C_2$ contains d_1.*

Trivial du-chains occur whenever a du-chain is completely determined by a suffix. E.g. a fun f is generated, freezing the definition of a variable v. An f-aware du-chain that covers a use of v in the body of f is trivial with respect to a corresponding set of non f-aware du-chains, if there is exactly *one* definition of v outside of f that can determine the frozen value.

The measurement results for the three test modules are presented in Table 3. For the measurements we have just considered serial test modules. We, therefore, do not take into account the message aware coverage criteria. We want to focus on some details of the measurement results that lead us to a proposition for data flow oriented testing of functional programs.

- Our measurements did not show any non-trivial a-aware du-chains with respect to the set of all du-pairs (i.e. all $\emptyset$-aware du-chains). In the further comparisons $\emptyset$-aware and a-aware du-chains behave identically.
  Nevertheless, a-awareness is the most simple extension to the set of all du-pairs. Missing a-awareness might cause overlooked correspondences of the other awareness levels. So it is not reasonable to extend the effort to cover other awareness levels, but to leave out a-awareness. All further du-chain sets are, hence, a-aware.

- The affect of freeze awareness, and structure awareness is quite small, when not considering result awareness. Since especially the additional effort for reaching this level is small, it is valuable to consider fs-awareness in order to get the additional strength in the appropriate cases.

- The most non-trivial du-chains are generated by the combination of structure awareness and result awareness. However, functions with many clauses, that

C_1	C_2	number of non-trivial chains in C_2 wrt. C_1		
		numbering_t	compute_throw_edges_t	gb_trees
$\emptyset$	a	0	0	0
$\emptyset$	af	8	0	0
$\emptyset$	as	6	0	0
$\emptyset$	ar	116	4	153
$\emptyset$	afs	14	0	0
$\emptyset$	afr	125	4	153
$\emptyset$	asr	5298	28037	214
$\emptyset$	afsr	7506	27979	214
a	af	8	0	0
a	as	6	0	0
a	ar	116	4	153
a	afs	14	0	0
a	afr	125	4	153
a	asr	5298	28037	214
a	afsr	7506	27979	214
af	afs	6	0	0
af	afr	118	4	153
af	afsr	7506	27979	214
as	afs	8	0	0
as	asr	5298	28037	214
as	afsr	7506	27979	214
ar	afr	9	0	0
ar	asr	5298	28037	61
ar	afsr	7506	27979	61
afs	afsr	7506	27979	214
afr	afsr	7506	27979	61
asr	afsr	2208	0	0

TABLE 3. Relation between data flow awareness levels

are called from many different places, can generate an infeasible number of sr-aware du-chains.[7]

A proposition taken from this measurements is the following. Test the asf-aware coverage criterion in every case. This awareness level does not increase the complexity of the testing process very much, but might lead to additional insight in

[7]The test modules *numbering_t*, and *compute_throw_edges_t* were modified here, in order to make at least the computation of all du-chains feasible. Both modules contain large functions, whose clauses make a case distinction on the next structure in the flow graph. Some of the non-recursive clauses of these functions had to be eliminated to make the computation of the measurement results possible.

some appropriate cases. It, therefore, offers a moderate gain at almost no costs. If the number of asfr-aware du-chains is feasible in the given testing context, test the asfr-aware criterion, which offers an even much higher gain whenever feasible.

6 CONCLUSION

Carrying over flow graph directed testing to a new programming paradigm consists of providing an adapted notion of flow graphs and the identification of coverage criteria that fit into the context of the new paradigm. While flow graphs for functional programming have already been described in the literature [Wid04], we have proposed a set of data flow oriented coverage criteria in this paper.

The data flow orientation gives us the opportunity to incorporate several ways of data flow. These are aliasing, the hiding of a value inside of a structure, the returning of values computed within a called function, and the freezing of values by the generation of a lambda closure, which is called at a distant place.

In the case of the functional language Erlang, there is an additional source of data flow, generated by the message passing mechanism between different processes. The data flow of these messages does not correspond to a control flow between the sender and the receiver. Therefore, data flow criteria are the only way to incorporate the distant relationships introduced by message passing.

While the proposed data flow oriented coverage criteria are more complex to check than simple node coverage (especially they rely on the computation of a flow graph), measurements show that even the simplest data flow oriented criterion contains significantly more information than node coverage.

The set of data flow criteria presented here contains both, simple and easy to check ones, and complex criteria, that can catch more bugs, but are also more complicated to be checked. Measurements for the different criteria lead to arguments that make the choice of the best data flow oriented criterion easy in a given context.

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.

[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.

[Eri03] Ericsson Utvecklings AB. *Erlang Reference Manual, Version 5.3*, 2003.

[ET] *Tools version 2.3*. Documentation of Erlang/OTP R9C.

[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.

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

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

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[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.

[Wid04] Manfred Widera. Flow graphs for testing sequential Erlang programs. In *Proceedings of the 3rd ACM SIGPLAN Erlang Workshop*. ACM Press, 2004.

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