Concurrent Erlang Flow Graphs

Manfred Widera
Fachbereich Informatik
FernUniversität in Hagen
58084 Hagen
Germany
manfred.widera@fernuni-hagen.de

ABSTRACT

Flow graphs are an important, and useful tool for testing programs or program components during software development. For imperative languages it is state of the art to use flow graph based coverage tools during the unit testing stage. Based on flow graphs for functional programming languages, that have to cope with higher order functions, a flow graph concept for Erlang needs a special treatment for the concurrent language constructs that are typical of Erlang. This paper presents a definition of flow graphs for Erlang programs that especially handles process generation and message passing, and describes how these flow graphs can be computed.

1. INTRODUCTION

Testing of software is a widely used method of detecting errors during the software development process. Every software is tested before being used 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 specification 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 Erlang, the only available implementation of source code directed testing is an ad hoc approach that checks the individual *lines* of a program for coverage [6]. 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 especially 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 Erlang 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.

The aim of this paper is to give a definition of flow graphs of 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. Based on an preliminary approach for sequential Erlang programs [23], the approach described here covers the whole Erlang standard, especially handling process generation and message passing.

The rest of the paper is organized as follows. In Sec. 2 related work is described and the current paper is classified in this context. Section 3 presents the language under consideration which is essentially given by the full Erlang standard without some of the syntactic sugar. A definition of concurrent flow graphs is given in Sec. 4. Section 5 recalls and refines some definitions of data flow analysis which are necessary for the computation of concurrent flow graphs presented in Sec. 6. Conclusions, and possible areas of future work are presented in Sec. 7.

2. RELATED WORK

2.1 Flow Graphs and Sequential Testing

The work presented here is related to publications from several areas. In imperative programming languages, flow graphs are accepted as a standard tool for checking test case coverage during the unit testing stage [24]. In the context of functional programming there are already approaches on flow graphs that are, however, not focused on test case coverage. Van den Berg [19] uses flow graphs, and call graphs for software measurement on 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.

A concept of generating flow graphs for higher order functional programs is described by Shivers [17] 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 [17]. 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 [4] 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 [14, 15, 16] 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. [8], [12], [3, 20]) 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 [6] implements a coverage test for Erlang modules, that analyzes the individual lines of the source code for coverage. It is, however, not able to distinguish between 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. Since the distant relationships between send operations, and receives in a program are also not considered by cover, the concurrent structure of Erlang programs is not an additional challenge for the system. In contrast, the preliminary approaches to flow graph based testing in sequential Erlang [23] need a nontrivial extension to handle concurrent Erlang constructs.

2.2 Testing Concurrent Programs

In testing concurrent programs, one is usually especially interested in certain interactions between the different processes or threads that lead to a number of specific errors like deadlocks and race conditions. The approaches to ensure the correctness of concurrent programs are divided into static and dynamic approaches.

Static analysis of concurrent programs is often done in the form of model checking. The underlying concept as well as the VeriSoft tool for performing model checking are described by Godefroid [9, 10].

The dynamic testing of concurrent systems is based on executing different schedules of synchronization events (i.e. events that are observable from outside the triggering process). Besides non-deterministic testing of schedules generated by chance, there are different systematic approaches for generating schedules and for enforcing their execution. Carver, and Tai [2] describe a deterministic testing approach by enriching the program code with special calls to a scheduler that is able to generate and repeat different schedules of interest. A similar approach of a scheduler function that is explicitly called is followed by Stoller [18] for Java Programs. Factor, Farchi, and Talmor [7] also address Java programs. Besides the schedule replay they focus on a coverage test for schedules. A further approach on systematically generating schedules is given by Hwang, Tai, and Huang [11]. For a given valid schedule new prefixes of schedules are generated by introducing minimal changes to the known schedules.

2.3 Classification of the Current Paper

The approach presented here stands in the tradition of testing sequential programs [24]: a flow graph oriented testing tool is applied to program parts that are usually too small for detecting the special forms of errors described in Subsec. 2.2. Concurrency and message passing are, however, very prominent parts of the Erlang design, such that a strategy in handling the concurrent language features to some extent by the structure oriented testing process is necessary. The aim of this work is to take into account the effects of message passing on the possible destinations of higher order function calls and on data flow oriented coverage criteria.

During the software development cycle the concurrent flow graphs described here, and the coverage criteria based on them can replace previously available structure oriented coverage approaches. Other stages of the testing process remain unchanged. This is especially the case for detecting synchronization errors, where tools as those described in Subsec. 2.2 can be employed.

3. PRELIMINARIES

The flow graph generation is defined (and is mostly implemented) for the whole Erlang standard [5]. The presentation in this paper is, however, restricted to 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.

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.

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 expression in P whose position is that of a boxed expression in Fig. 1 consists of an instantiated variable.*

constants a: $\star$

variables X: $\star$

patterns p: $a|X|\{p_1,\ldots,p_k\}|[p_1|p_2]|[p_1,\ldots,p_k]$

guards g: $\star$

if clauses ic: $g \rightarrow l$

case clauses cc: p [when g] $\rightarrow l$

fun clauses fc: $(p_1,\ldots,p_k)$ [when g] $\rightarrow l$

function name fn: $\star$

expressions e: $a|X|\boxed{e_0}(\boxed{e_1},\boxed{e_2},\ldots,\boxed{e_k})|fn(\boxed{e_1},\boxed{e_2},\ldots,\boxed{e_k})|p = e|\{\boxed{e_1},\ldots,\boxed{e_k}\}|[\boxed{e_1}|\boxed{e_2}]|$
$[\boxed{e_1},\ldots,\boxed{e_k}]|$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$(\boxed{e})$|send$(\boxed{e},\boxed{e})$|receive $cc_1; cc_2; \ldots; cc_k$ end

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

- *Each function consists of a single clause with just variables as arguments.*[1]
- *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 useful for performing data flow analysis and for presenting the data flow results.

4. CONCURRENT FLOW GRAPHS

The definition of concurrent flow graphs is given in two stages. Subsection 4.1 defines the basic properties of a concurrent flow graph. The correspondence between a program P and its flow graph V_P is defined in Subsec. 4.2.

4.1 Basic Flow Graph Definition

As ordinary flow graphs known from literature [24], concurrent flow graphs are given by sets of nodes and edges. These sets are partitioned into a number of subsets. Their definition is given in the following Definitions 2 and 3.

Essentially, each expression in a program is represented by an individual node.[2] These nodes are labeled to express all information given by the expression itself. In order to assign specific labels for the different kinds of expressions, different kinds of nodes are necessary.

[1]To enforce this, the case distinction of the different function clauses, and the value decomposition by their patterns need to be performed by a case-expression inside the single clause. For functions with arity > 1 the arguments and the corresponding patterns are structured into tuples of equal element number.

[2]This also holds in the case of nested sub-expressions, which have been eliminated by the preprocessing stage enforcing the named definition property.

DEFINITION 2 (NODES). *The set V of nodes of a concurrent flow graph is divided into the following subsets.*

- $V_{match} \subset V$ *denotes the set of all match nodes. A match node is labeled by a pattern LHS and a further node $RHS \in V$.*
- $V_{call} \subset V$ *denotes the set of all call nodes. A call node is labeled with a function call $e_0(e_1,\ldots e_k)$ or $fn(e_1,\ldots e_k)$. Each call node occurs as label of a match node in the flow graph.*
- $V_{spawn} \subset V$ *denotes the set of all spawn nodes. A spawn node is labeled with a function (given by module name and function name) and a sequence $e_1,\ldots,e_k$ of argument expressions.*
- $V_{branch} \subset V$ *denotes the set of all branching nodes. A branching node is labeled with a sequence $e_1,\ldots e_k$ of $k \geq 0$ tests, and for each branch with a sequence of k patterns, and a set of guards.*
- $V_{block} \subset V$ *denotes the set of all block nodes. A block node is labeled with a set of nodes.*
- $V_{catch} \subset V$ *denotes the set of catch nodes. A catch node is labeled with a further concurrent flow graph node n.*
- $V_{throw} \subset V$ *denotes the set of throw nodes. A throw node is labeled with an expression e.*
- $V_{send} \subset V$ *denotes the set of send nodes. A send node is labeled with two expressions, the* destination expression e_d *and the* message expression e_m.
- $V_{receive} \subset V$ *denotes the set of receive nodes. A receive node contains a set of branches, and is labeled with a pattern, and a sequence of guards for each branch.*

- $V_{fun} \subset V$ *denotes the set of fun nodes. A fun node is labeled with a function name and an arity.*[3]
- $V_{import} \subset V$ *denotes the set of import nodes. An import node is labeled with a list of variables.*
- $V_{context} \subset V$ *denotes the set of context nodes. A context node is labeled with a set of pairs* $(Var, Defs)$ *where* Var *is a variable* v *and* $Defs$ *is a list of references to nodes potentially assigning a value to* v*.*
- $V_{return} \subset V$ *denotes the set of return nodes. A return node is labeled with a variable.*
- $V_{compute} = V \setminus (V_{match} \cup V_{call} \cup V_{spawn} \cup V_{branch} \cup V_{block} \cup V_{catch} \cup V_{throw} \cup V_{send} \cup V_{receive} \cup V_{fun} \cup V_{import} \cup V_{context} \cup V_{return})$ *denotes the set of all computation nodes. Each node* $n \in V_{compute}$ *is labeled with an expression, that is not a match, a call, a branch (i.e. if, case), a begin, a catch, a throw, a send, a receive, or a fun.*

All described subsets of V *are pairwise disjoint.*

For concurrent flow graphs several different kinds of edges are necessary. Usually an edge in a flow graph describes the (directed) control flow, and data flow between two nodes. In concurrent flow graphs we distinguish two kinds of edges with this property. *Neighborhood edges* connect nodes whose represented expressions are adjacent in the source code. The non-local returns given by the catch-throw mechanism in Erlang are expressed by *throw edges* in concurrent flow graphs.

Call edges express function calls, and are special in the sense that they are bidirectional and represent both the control and data flow during a function call, *and* during the return from the call. (For the generation, we distinguish first order call edges and higher order call edges, depending on the call represented by their source node.)

Two further forms of edges just represent a data flow, but no control flow. *Spawn edges* essentially represent the data flow during the process generation. They do not represent an ordinary control flow, because the new process generated by them forms a new independent instance of control. *Message edges* finally stand for the data flow performed by the message passing mechanism between a send expression and a receive expression. No control flow occurs between the processes connected by a message edge.

The formal definitions of these kinds of edges are given by the following Def. 3.

Definition 3 (edges). *The set* E *of edges of a concurrent flow graph is divided into the following subsets.*

- $E_{call} \subset E$ *denotes the set of call edges. Source of a call edge is a call node* $n_s \in V_{call}$ *with call arguments* $a_1, \ldots a_k$*; destination is an import node* $n_d \in V_{import}$ *with parameters* $p_1, \ldots, p_n$ *such that* $k = n$*. A call edge is labeled with the following information:*
  - *A set of assignments* $p_i = a_i$ *for* $i \in \{1, \ldots, k\}$ *called* parameter assignments.
  - *An assignment* $u = r$ *where* r *is the return variable of the function starting with* n_d*, and* u *is the pattern, occurring besides* n_s *as a label of a match node. This is called the* result assignment.

  Call edges are divided into first order call edges and higher order call edges, depending on the call represented by their source call node.
- $E_{spawn} \subset E$ *denotes the set of all spawn edges. Source of a spawn edge is a spawn node* $n_s \in V_{spawn}$ *labeled with the function expression* f *and the argument expressions* $e_1, \ldots, e_k$*; destination is the import node* $n_d \in V_{import}$ *of a function with parameters* $p_1, \ldots, p_n$ *fulfilling* $k = n$ *such that there is at least one execution of the containing program* P *with* f *denoting the destination function. A spawn edge is labeled with a set of parameter assignments* $p_i = e_i$ *for* $i \in \{1, \ldots, k\}$*.*
- $E_{throw} \subset E$ *denotes the set of all throw edges. A throw edge has a throw node as source, and a catch node as destination.*
- $E_{message} \subset E$ *denotes the set of all message edges. A message edge runs from a send node to a receive node.*
- $E_{neighbor} = E \setminus (E_{call} \cup E_{spawn} \cup E_{throw} \cup E_{message})$ *denotes the set of neighborhood edges. Neighborhood edges emerging from a branching node or a receive node are labeled with a clause number.*

The described subsets of E *are pairwise disjoint.*

Combining the definitions of nodes and edges, we get the following definition of a concurrent flow graph.

Definition 4 (concurrent flow graph). *A concurrent flow graph is a pair* $G = (V, E)$ *where*

- *The set* V *is divided into subsets* V_{match}*,* V_{call}*,* V_{spawn}*,* V_{branch}*,* V_{block}*,* V_{catch}*,* V_{throw}*,* V_{send}*,* $V_{receive}$*,* V_{fun}*,* V_{import}*,* $V_{context}$*,* V_{return}*, and* $V_{compute}$ *according to Def. 2.*
- *The set* E *of edges is divided into the subsets* E_{call}*,* E_{throw}*,* $E_{message}$*, and* $E_{neighbor}$ *according to Def. 3.*

Example 1 (concurrent flow graph). *A graphical example of a concurrent flow graph is given in Fig. 2. For simplicity reasons, the presentation is simplified as follows.*

- *The context nodes are empty for all the functions and are omitted in the graphical representation.*
- *Match nodes are marked with dotted lines inside the nodes they are labeled with.*

Call and spawn edges are marked by rounded corners. Receive nodes are marked by a triangle which is connected to a number of rows, each containing the pattern and the guards

[3]Note that a fun node does not correspond to a fun expression directly: the node is labeled with a name for the function instead of the function's clauses.

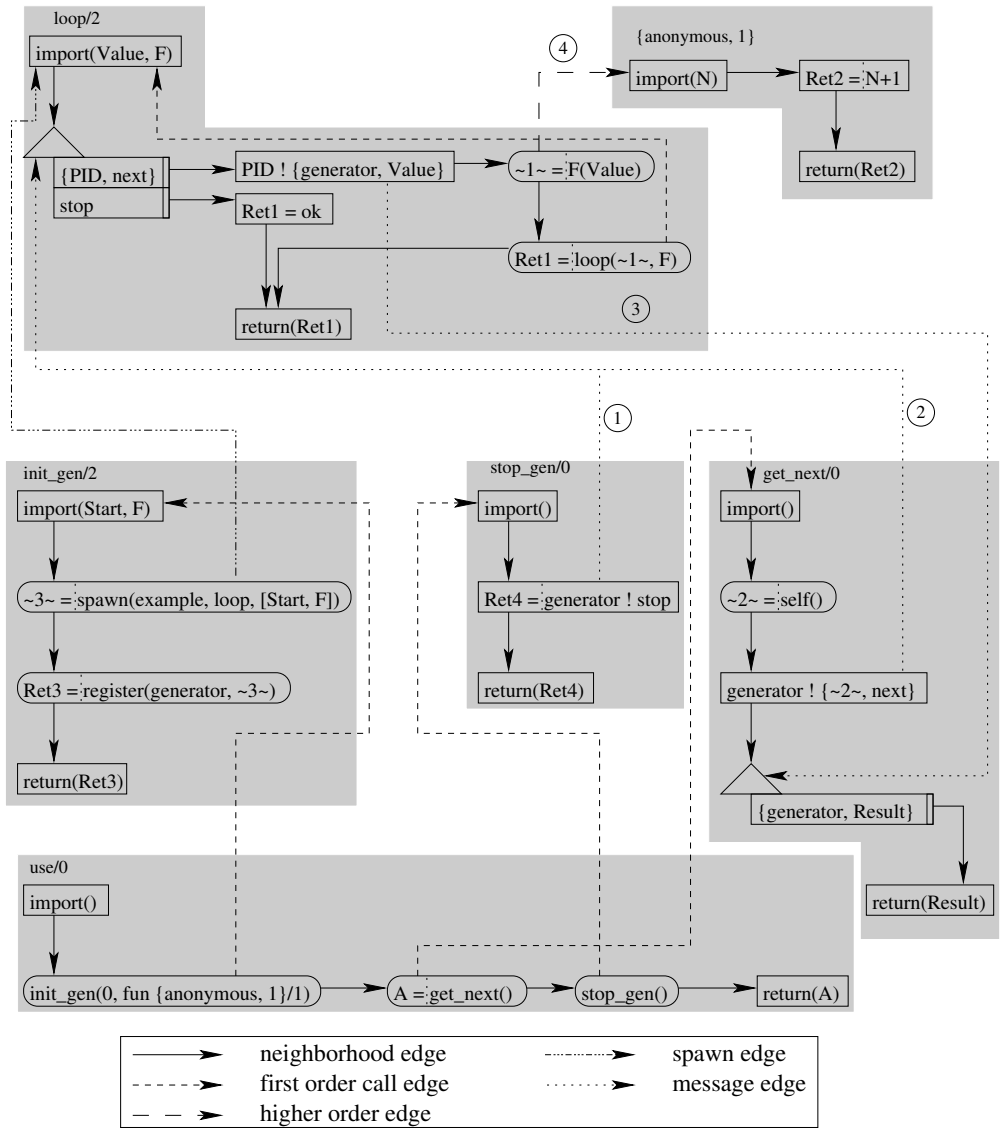

Figure 2: Flow graph of example.erl

for one clause. In order to simplify the identification of functions, the nodes of each function are contained in a gray box.

The numbers marking some of the edges are not of interest for the moment. They will be used later for describing the iterated edge computation by our implementation.

The graphical representation of the node forms not occurring in Ex. 1 is identical to sequential Erlang flow graphs [23].

4.2 The Concurrent Flow Graph of a Program

Given an Erlang program P fulfilling the named definition property, the following Definitions 5, 6, and 7 describe the concurrent flow graph G_P corresponding to P, i.e. the flow graph that can be used to represent P.

Note that the definition of G_P given here is not intended to provide an algorithm for computing G_P. Indeed, an algorithm will just be able to compute an approximation of G_P instead of G_P itself. The presentation of the implementation in Sec. 6 will discuss sources of inaccuracy in the computation, and the effect of the necessary approximations on the higher order call edges, spawn edges, throw edges, and message edges.

We start the presentations with the correspondence between the nodes in G_P, and the program expressions in P. Essentially, for each expression in the program a node is generated. The kind of node chosen depends on the structure of the expression. Additionally, each function definition is extended by an import node representing a local definition of the parameters of the function, a context node representing local definitions of the variables taken from the context of

the function (this just applies to funs), and a return node representing the return to the calling program part.

DEFINITION 5 (CORRESPONDING NODES). *Let P be a program fulfilling the named definition property. The set V_P of corresponding nodes for P is generated by the following rules.*

- *For each expression e denoting a function call of the form $\mathtt{e_0(e_1,\ldots e_k)}$ or $\mathtt{fn(e_1,\ldots e_k)}$ V_P contains a call node $v \in V_P$ labeled with e.*
  *In the special case of a call $\mathtt{fn(e_1,\ldots e_k)}$ with $\mathtt{fn}$ denoting the BIF **spawn/3** or **spawn/4**, a spawn node is generated instead, which is labeled with the function and the call arguments given by the arguments of **spawn** at the corresponding positions.*
- *For each expression of the form $\mathtt{if\ ic_1;\ldots;ic_k}$ there exists a branching node $n \in V_P$ which is labeled with the guards of the individual clauses.*
- *For each expression of the form $\mathtt{case\ e\ of\ cc_1;\ldots;cc_k}$ there exists a branching node $n \in V_P$ which is labeled with the test e, and with patterns, and guards of the individual clauses.*
- *For expressions of the form $\mathtt{begin\ l\ end}$ a block node is introduced which is labeled with the nodes generated for the expressions in l.*
- *For each expression of the form $\mathtt{catch(e)}$ a catch node is generated and labeled with the node for the subexpression e.*
- *For each expression of the form $\mathtt{throw(e)}$ a throw node is generated and labeled with the subexpression e.*
- *For each expression of the form $\mathtt{send(e_1,e_2)}$ a send node is generated and labeled with e_1 as destination expression and e_2 as message expression.*
- *For each expression of the form $\mathtt{receive\ cc_1;\ldots;cc_k}$ there exists a receive node $n \in V_P$ which is labeled with the patterns and guards of the individual clauses.*
- *For every function definition in P and every expression of the form $\mathtt{fun\ fc_1;\ldots;fc_k}$ the following nodes are in V_P.*
  - *An import node labeled with the formal parameters of the function.*
  - *A context node labeled with all variables v, and references to the defining nodes n such that v is defined outside the function,[4] and the definition of v in n reaches a use within the function.*
  - *A branching node labeled with the patterns and guards of the individual function clauses.*
  - *A return node labeled with the return variable of the function.*

  For expressions of the form $\mathtt{fun\ fc_1;\ldots;fc_k}$ a fun node is generated which is labeled with a generated function name and the function arity.
- *For every expression e of the form $\mathtt{p = e'}$ there is a match node $v \in V_P$ labeled with the pattern p and the node v' generated for e'.*
- *For each expression e of the form $\mathtt{a}$, $\mathtt{X}$, $\{\mathtt{e_1},\ldots,\mathtt{e_k}\}$, $[\mathtt{e_1}|\mathtt{e_2}]$, or $[\mathtt{e_1},\ldots,\mathtt{e_k}]$ a computation node is generated and labeled with the expression e.*

Edges represent a control or data flow between the individual nodes. Their definition is based on the runtime behaviour of the program (which will be approximated for the computation of flow graphs).

DEFINITION 6 (CORRESPONDING EDGES). *Let P be a program fulfilling the named definition property, and V_P the set of corresponding nodes for P.*
Now let $n_1, n_2 \in V_P$ be nodes, and let e_1 and e_2 be the expressions in P the nodes n_1 and n_2 correspond to, respectively.[5] *The set E_P of corresponding edges for P consists of all edges generated by one of the following rules.*

- *There exists a neighborhood edge from n_1 to n_2 in E_P if e_1 and e_2 belong to the same function f, and one of the following conditions holds.*
  - *n_1 is the import node, and n_2 the context node of f.*
  - *n_1 is the context node, and e_2 the first expression of f.*
  - *e_2 is the direct successor of e_1 in a sequence of expressions.*
  - *n_1 is a branching node or a receive node, and e_2 is the first expression in one of the clauses belonging to e_1. In this case the edge is labeled with the clause, e_2 belongs to.*
  - *e_1 is the last expression in a clause, and e_2 is the expression following the if, case or receive expression containing e_1.*
  - *e_1 is the last expression of one of the clauses of f (if the last expression e' is an if, case or receive expression, e_1 is the last expression in the body of one of the clauses of e'), and n_2 is the return node of f.*
- *There exists a call edge from n_1 to n_2 in E_P if n_1 is a call node, n_2 is the import node of a function f, and there exists an execution of P such that e_1 performs a call to f.*
- *There exists a spawn edge from n_1 to n_2 in E_P if e_1 is a spawn node, e_2 is the import node of a function f, and there exists an execution of P such that e_1 spawns a process starting its execution with f.*
- *There exists a throw edge from n_1 to n_2 in E_P if n_1 is a throw node, n_2 is a catch node, and there exists an execution of P such that e_1 throws a value that is caught by e_2.*

[4] This only applies to funs. For named functions the context node is empty.

[5] If n_i is an import node, a context node, or a return node then e_i is undefined.

```
-module(example).
-export([loop/2, use/0]).

loop(Value, F) ->
    receive
        {PID, next} ->
            PID ! {generator, Value},
            loop(F(Value), F);
        stop -> ok
    end.

init_gen(Start, F) ->
    register(
        generator,
        spawn(example, loop, [Start, F])).

stop_gen() -> generator !  stop.

get_next() ->
    generator !  {self(), next},
        receive
            {generator, Result} -> Result
        end.

use() ->
    init_gen(0, fun(N) -> N + 1 end),
    A = get_next(),
    stop_gen(),
    A.
```

Figure 3: Erlang source code of example module

- *There exists a message edge from n_1 to n_2 in E_P if n_1 is a send node, n_2 is a receive node, and there exists an execution of P such that e_1 sends a message that is received by e_2.*

Combining corresponding nodes and corresponding edges of a program P, we get the concurrent flow graph G_P of P.

Definition 7 (corresponding flow graph). *Let P be a program fulfilling the named definition property. The corresponding flow graph for P is defined by $G_P = (V_P, E_P)$ where V_P is the set of corresponding nodes for P and E_P is the set of corresponding edges for P.*

Example 2 (corresponding flow graph). *Consider the module `example.erl` that is presented in Fig. 3, and that contains the following functions.*

- *`loop/2` forms a sequence generator that is meant to reside in an own process. It is initialized with the initial value, and the successor function of the sequence. For every message `{PID,next}` it sends the next sequence element to the process `PID`.*
- *`init_gen/2`, `stop_gen/0`, and `get_next/0` are the accessor functions for initializing, stopping and accessing the generator process.*
- *`use/0` is an example user of the sequence generator. It initializes the generator with the sequence of all natural numbers starting from 0, queries the first sequence element, stops the generator process and returns the element.*

The concurrent flow graph corresponding to `example.erl` is the one presented in Fig. 2.

5. DATA FLOW ANALYSIS

As stated by Shivers [17], 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 give a definition (adapted towards the use for concurrent flow graphs) of some base notions of data flow analysis, that are known for imperative languages [13]. For a definition of a variable v we write $def(v)$, for a use of v we write $use(v)$. The precise definitions of these notions are as follows.

Definition 8 (definitions). *Let G be a concurrent 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 an f-definition (denoted by f-def(v)).*
- *n is a match node denoting a 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 is a branching node, 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.*
- *n is a receive node, 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. This is called an m-definition (denoted by m-def(v)).*

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 9 (uses). *Let G be a concurrent 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 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 branching node with a test given by v.*
- *n is a fun node, 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 is a match node denoting a 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 branching node, or a receive node, 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.*
- *n is a send node and v occurs either as destination expression or as message expression. If v is the message expression, this is called an m-use and denoted by m-use(v).*

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

For the pairs, f-definition/f-use, s-definition/s-use, and m-definition/m-use we need to define the notion of *corresponding* uses and definitions: each of these kinds of use can *hide* a value from the data flow analysis. The corresponding definition makes the value available again (possibly under a different name).

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

The precise definitions are as follows.

DEFINITION 10 (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 11 (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 there exists at least on run of the containing program P such that the structure decomposed in d is c, and the selected element position is the one containing the value of v.*

DEFINITION 12 (CORRESPONDING *m-use*, *m-def*). *Let v be a variable, and u an m-use of v in a node n_u. An m-definition d of some v' in a node n_d corresponds to u if there is a message edge from n_u to n_d in the concurrent flow graph.*

Note that for an s-use, and the corresponding s-definition or for an m-use, and the corresponding m-definition the variable names usually differ.

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

DEFINITION 13. *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 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 some $\tilde{v}$ and there is a corresponding f-definition of $\tilde{v}$ that reaches u.*
- *d reaches an s-use of v and there is a corresponding s-definition of some $\tilde{v}$ that reaches u.*
- *d reaches an m-use of v and there is a corresponding m-definition of some $\tilde{v}$ that reaches u.*

Besides the data flow coded in some of the nodes, the *labels of edges* can contain data flow information. This is the case for the parameter assignments given by call edges and spawn edges, and the result assignments of the call edges. These labels have to be taken into account for the data flow analysis. They are processed analogously to a sequence of nodes containing copy expressions when the edge is followed in the corresponding direction.

6. COMPUTATION OF CONCURRENT FLOW GRAPHS

For a program P fulfilling the named definition property the generation of the concurrent flow graph G_P consists of the following stages.

1. Generation of the set V_P of nodes according to Def. 5.

2. Computation of the set E_{neighbor} of neighborhood edges.

3. Computation of the call edges for first order function calls.

4. Computation of the call edges for higher order calls, the throw edges, and the message edges by an iterated process.[6]

Step (4) contains the computation of all edges that depend on the data flow in the program. It is necessary to iterate over all these edges because each new edge adds new opportunities for data flow in the graph.

The steps (1), (2), and (3) consist of a direct transfer of the corresponding definitions. They can be implemented in a straightforward manner. A detailed description of Step (4) is given in the following Subsec. 6.1.

6.1 Iterated Edge Computation

The generation of edges is implemented in form of three functions.

1. The computation of higher order call edges is done by a function

$$\texttt{ho\_call\_edges}(\texttt{Graph1}) \rightarrow \{\texttt{Graph2}, \texttt{Bool}\}$$

2. For computing the throw edges in the concurrent flow graph, we use a function

$$\texttt{throw\_edges}(\texttt{Graph1}) \rightarrow \{\texttt{Graph2}, \texttt{Bool}\}$$

3. The computation of the message edges is done by a function

$$\texttt{message\_edges}(\texttt{Graph1}, \texttt{Process}) \rightarrow \{\texttt{Graph2}, \texttt{Bool}\}$$

Each function expects a flow graph as input and returns a tuple containing a new flow graph and a boolean flag whether any new edges have been introduces. For the computation of the message edges in `message_edges/2` a description of the initial process, essentially given by its initial call (or a list of potential initial calls), must be provided as additional argument.

The main loop `introduce_edges/2`, which is presented in Fig. 4, loops over the three functions until no change was made by any of them in one step.

In the remaining presentation of `ho_call_edges/1`, `throw_edges/1`, and `message_edges/2` we omit the boolean flag for changes in the return value in order to simplify the presentation. In the following, the high level structure of the functions is presented, but we omit several of the called functions. The names of the omitted functions are chosen to represent their general semantics.

[6]Spawn edges behave very similar to higher order call edges during the computation. To simplify the following presentation, spawn edges are not discussed explicitely. Their creation is done together with the higher order call edges in an analogous manner.

```
introduce_edges(Graph, InitProcess) ->
   % generate new edges
   {GraphWithCall, CallChangeFlag} =
      ho_call_edges(Graph),
   {GraphWithThrow, ThrowChangeFlag} =
      throw_edges(GraphWithCall),
   {GraphWithMessage, MessageChangeFlag} =
      message_edges(GraphWithThrow, InitProcess),
   % check whether a further step is necessary
   if
      CallChangeFlag;
      ThrowChangeFlag;
      MessageChangeFlag ->
          % next iteration with new graph
          introduce_edges(
              GraphWithMessage, InitProcess);
      true ->
          % return graph with computed edges
          GraphWithMessage
   end. % if
```

Figure 4: computation of data flow dependent edges

- `ho_call_edges/1` is presented in Fig. 5. For each higher order function call it analyzes the variable in the function position. For each reaching definition which denotes a function within the graph,[7] a call edge is introduced.

- `throw_edges/1` is presented in Fig. 6. For each catch node c, it determines the set of throw nodes t such that there exists a path from c to t without another catch node in-between. For each such t a throw edge from t to c is introduced.

- `message_edges/1` is presented in Fig. 7. First, it calculates the set of processes from the initial process. This is done by an iterated analysis of the spawn nodes reachable from the already known processes. For each send node s in the graph, the following steps are performed afterwards.

  1. The first argument of the send node is analyzed to determine the potential destination processes from the set of all processes. This is done by data flow analysis and partial evaluation on the reaching definitions.
  2. The variable v in the second argument of the send node is tested against the receive statements of the destination processes: if one of the definitions reaching the use of v in the send node matches one of the patterns of the receive r then a message edge from s to r is inserted into the graph.

EXAMPLE 3. *We reconsider the module* `example.erl` *presented in Fig. 3, and its corresponding concurrent flow graph*

[7]This property is checked using partial evaluation.

```
ho_call_edges(Graph) ->
     % extract higher order calls
     Sources = ho_calls(Graph),
     % compute and insert edges for each call independently
     foldl(fun(Call, GraphAcc) -> edges_from_call(Call, GraphAcc) end, Graph, Sources).

edges_from_call(Call, Graph) ->
    % compute all reaching definitions of function position in call that denote a function
    Dest = filter(fun is_function/1, reaching_definitions(extract_called_function(Call))),
    % insert edges from Call to each element of Dest into Graph
    foldl(fun(SingleDest, GraphAcc) -> insert_call_edge(Call, SingleDest, GraphAcc) end, Graph, Dest).
```

Figure 5: Computation of higher order call edges

```
throw_edges(Graph) ->
    % extract catch nodes from graph
    Dest = catch_nodes(Graph),
    % compute and insert the throw edges for each catch node independently
    foldl(fun(Catch, GraphAcc) -> edges_to_catch(Catch, GraphAcc) end, Graph, Dest).

edges_to_catch(Catch, Graph) ->% compute all throw nodes that are reachable from the current
                               % catch without a further catch on the path
    Source = filter(fun(PossibleSource) ->
                            % source is throw node, no other catch between Catch and trow node
                            is_throw_node(PossibleSource)
                            and not(catch_on_each_path(Catch, PossibleSource))
                    end, nodes_reached_from(Catch)),
    % insert an edge from each element of Source to Catch into Graph
    foldl(fun(SingleSrc, GraphAcc) -> insert_throw_edge(SingleSrc, Catch, GraphAcc) end, Graph, Source).
```

Figure 6: Computation of throw edges

```
message_edges(Graph, InitProcess) ->
    % compute processes starting from InitProcess
    Processes = compute_all_processes(Graph, InitProcess),
    % extract send nodes from graph
    Source = send_nodes(Graph),
    % compute and insert the throw edges for each send node independently
    foldl(fun(Send, GraphAcc) -> edges_from_send(Send, GraphAcc, Processes) end, Graph, Source).

edges_from_send(Snd, Graph, Processes) ->
    % compute all processes that can be the destination of the current send
    DestProc = filter(fun(PossibleDest) -> member(PossibleDest, Processes) end,
                      reaching_definitions(extract_send_destination(Snd))),
    % compute set of sent messages
    Messages = reaching_definitions(extract_send_message(Snd)),
    % compute all receives in all destination processes that match an element of Messages
    Dest = filter(fun(Receive) -> receive_matches_value(Receive, Messages) end,
                      % test all receives of all destination processes
                      append(map(fun get_process_receive_nodes/1, DestProc)))),
    % insert edges from Snd to each element of Dest into Graph
    foldl(fun(SingleDest, GraphAcc) -> insert_message_edge(Snd, SingleDest, GraphAcc) end, Graph, Dest).
```

Figure 7: Computation of message edges

presented in Fig. 2. The graph generation in this case consists of the following steps.

1. *The nodes, the neighborhood edges, and the first order call edges (including the spawn edge) are generated.*
2. *During the first iteration of edge generation the following edges are introduced.*
   - *The higher order call edge marked with* 4.
   - *The message edges marked with* 1 *and* 2.
3. *During the second iteration the message edge marked with* 3 *is introduced. It is delayed to this step, because it makes use of the data flow of variable* `~2~` *to* `PID` *on the message edge marked with* 2.
4. *The third iteration does not introduce any further edges. Therefore the iteration process terminates.*

6.2 Sources of Inaccuracy

The algorithm presented previously in this section cannot compute the concurrent flow graph according to Def. 7 exactly because of a number of sources of inaccuracy during the computation. All these inaccuracies just affect the sets of higher order call edges, throw edges, and message edges. The computation of the nodes, the neighborhood edges, and the first order call edges does not depend on the data flow analysis, and can be done in a precise manner.

In detail the data flow analysis is affected by the following effects.

- Distant conditionals in a program can correspond in a way, that only certain combinations of subpaths can occur in a path. For instance, consider two functions f, g that are called with the same argument n, and both contain different branches for even and for odd values of n. A path through the program that takes the even branch in f but the odd branch in g for the same n is not possible. Such situations are not recognized by the system.
- The control flow is based on 0CFA [17], i.e. different function closures sharing the same source code are identified. When distinguishing these functions, several nodes are necessary for what is represented by one node in our approach. Especially, for higher order function calls, each of these nodes could be more specific in the sense of having less outgoing call edges than the one node being the source for the union of all these call edges in our concurrent flow graphs.
- In Erlang the function position of a higher order function call can be given by a fun, or a representation of the *name* of the called function. The second case is problematic because partial evaluation is necessary to identify the called functions. This partial evaluation must, however, be approximate, especially if the needed information is not completely available before runtime. (Example: the name of a called function is read from the keyboard at runtime.)
- Parts of the data flow can escape the scope of the flow graph. In that case approximating assumptions must be used. For example this is the case when a function f is passed to a function g as argument, but g is not part of the code under testing. In this case we cannot be sure whether f is called from g, or not.
- The test whether a definition matches a pattern of a receive statement must approximate the value of the definition. Therefore the computation of the message edge destinations is approximate.

Approximation is done according to the following policy. Whenever there is any doubt whether an edge is necessary, the edge is introduced. This guarantees the concurrent flow graph to contain as much control and data flow alternatives as possible.

If there is, however, no information about the alternatives at a certain point (e.g. when reading a name of the function to be called from the keyboard) we do not insert edges to all possible destinations (e.g. to all functions of the right arity occurring in the flow graph).

6.3 Implementation

The computation of concurrent flow graphs for Erlang programs is implemented based on OTP R9C-2, i.e. both the implemented code, and the programming constructs expected in the analyzed program are based on this language standard.[8]

The flow graph structure is essentially based on the parse result of the OTP library module `epp.erl`. This module is used to read the source code modules under testing, and the result format is preserved during the preprocessing stage enforcing the named definition property. The flow graph consists of a list of modules, each given by the result of `epp:parse_file/3` which is modified in the following steps to form the flow graph.

The node generation consists of a traversal of the code performing the following tasks.

- In each expression representing a node, the line number entry of the `epp` output is changed to a tuple additionally holding a node number, and some local data flow information.
- For each fun generation consisting of function clauses, a new function name is generated, which replaces the clauses in the definition. The clauses are taken to represent the generated function in the flow graph.
- Some of the structures denoting special expressions are replaced or extended.
  - Calls to the BIF `throw/1` are replaced by a new kind of tuple structure.

[8] At the moment, the implementation lacks the handling of list comprehensions, that are a bit tedious to cope with, but do not provide any new problems or insight. Code for handling them will be added, once the restricted prototype is finished.

  - Entries for function calls are extended by a field for the call edge information.
  - Receive entries are extended by a field for information on the message edges.

- For each function, the additional import, context and return nodes are introduced, which are represented by tuples similar to those returned by `epp`.

Some further information is pre-calculated and stored for future use. Among others, a list of all call nodes (and other important node types) is stored for each function, and an index given by a balanced tree is generated for each module for accessing the individual nodes by their number.

In a next step, the call nodes are divided into first order calls and higher order calls. For the first order calls, the destinations can be computed easily, and the resulting edge information is stored in the prepared field of the call.

For the higher order call edges, the spawn edges, the throw edges, and the message edges, the computation is done as described in Subsec. 6.1, and the computation results are coded into the prepared fields of the structures.

7. CONCLUSION AND FUTURE WORK

By adapting the notion of flow graphs to functional programs written in Erlang, especially containing the concurrent constructs integrated in the Erlang standard, 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.

As already stated for flow graphs for sequential Erlang [23], function calls have a strong influence on the control flow in functional programs comparable to the looping constructs in imperative languages. A refined definition of call edges is given here, providing a notion of expressing the whole 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.

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 [23]. Throw edges and message edges depend (like higher order call edges and spawn edges) on the possible control flow in the program and therefore on the computed higher order call edges. They, however, cause additional data flow opportunities and can therefore extend the set of higher order call edges in a program. An iteration looping over the generation of higher order call edges (with spawn edges), throw edges and message edges has been described that computes all these edges, and terminates when a fixpoint is reached.

Future work towards a coverage system based on concurrent flow graphs consists of two steps. First, a tracing tool storing the control flow through the tested modules during a test case execution [22] must be extended to handle several processes, and to store information on the data flow generated by passing messages between tested modules. As a second step, a tool analyzing given traces for their coverage level must be implemented. While a simple node coverage test is already finished, we expect data flow oriented coverage [21] to be of special use in the context of concurrent Erlang programs, because data flow coverage is the only level of coverage analysis that is able to consider messages passed around the program.

8. 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] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.

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

[4] 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.

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

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

[7] M. Factor, E. Farchi, and Y. Talmor. Timing dependent bugs. In *Software Testing Analysis and Review (STAR98)*, May 1998.

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

[9] P. Godefroid. Model checking for programming languages using VeriSoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[10] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2), Mar. 2005.

[11] G.-H. Hwang, K.-C. Tai, and T.-L. Huang. Reachability testing: An approach to testing concurrent software. In *Proc. First Asia-Pacific Software Engineering Conference (APSEC)*, Tokyo, Japan, Dec. 1994.

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

[13] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] S. D. Stoller. Testing concurrent java programs using randomized scheduling. In *Proceedings of the Second Workshop on Runtime Verification (RV)*, volume 70 of *Electronic Notes in Theoretical Computer Science*, 2002.

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

[20] 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.

[21] M. Widera. Data flow considerations for source code directed testing of functional programs. In H.-W. Loidl, editor, *Draft Proceedings of the Fifth Symposium on Trends in Functional Programming*, Nov. 2004.

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

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

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