# Flow Graph Interpretation for Source Code Directed Testing of Functional Programs

Manfred Widera

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

**Abstract.** Testing of software components during their development is a heavily used approach to detect programming errors and to evaluate the quality of software. Systematic approaches to software testing get a more and more increasing impact on software development processes.
For imperative programs there are several approaches to measure the appropriateness of a set of test cases for a program part under testing. Some of them are source code directed and are given as coverage criteria on flow graphs.
This paper sketches a tool for source code directed test analysis for the functional language Erlang. It motivates the need for an interpreter for flow graphs, and describes implementation and properties of such an interpreter.

## 1 Introduction

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

In the context of functional programming, there just exist simple ad hoc approaches to source code directed testing, as e.g. the cover tool for Erlang [1] that checks the individual *lines* of a program for coverage. As systematic testing is an important task of professional software development, it is desirable to have more advanced source code oriented testing methods for functional programming languages available.

In this paper, a system for source code directed testing of programs in the language Erlang is sketched. The focus of the paper lies on a prototype implementation of an interpreter for flow graphs that forms one of the central components of the testing tool. Besides the description of the basic design decisions and the implementation of the interpreter, the paper contains several measurement results for the interpreter compared with the Erlang runtime system.

The rest of the paper is organized as follows. Section 2 contains a discussion of related work. In Sec. 3 an overview over usual testing procedures, and over the considered Erlang subset are presented as preliminaries. Section 4 describes the top level structure of a source code directed testing tool for Erlang programs, and in Sec. 5 the implementation of a graph interpreter forming one core component of the testing tool is described. Section 6 describes some properties of the graph interpreter by presenting and discussing several measurement results. Finally, the conclusions are drawn and directions of future work are shown in Sec. 7.

## 2 Related Work

Flow graphs in functional programming, and the approach to use them for testing functional programs are related to publications from several areas. There are already approaches on flow graphs for functional languages. Van den Berg [2] uses flow graphs and call graphs in the context of software measurement for functional programs. The flow graphs used there consider function calls as atomic operations and are generated for each function independently. Information on calls between functions is given by a call graph as separate structure.

The concepts of generating flow graphs for higher order programs is described by Shivers [3] and further analyzed by Ashley/Dybvig [4]. 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 [3]. 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 [5] 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 [6–8] 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. [9], [10], [11, 12]) 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 [1] implements a coverage test for Erlang source code that analyzes the individual lines of the source code for coverage. It does not use an interpreter, but compiles the modules to be analyzed in a special way. Cover is, however, not able to distinguish several computations coded within a single line, or to check non local relationships e.g. between calls and called functions or between throws and corresponding catches.

## 3 Preliminaries

### 3.1 An Overview over the Testing of Software

Literature on testing imperative programs (e.g. [13]) usually distinguishes three different stages of testing during the software development process.

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

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

For *integration testing* several units (that have been tested individually during component testing) are connected to each other, and their interaction is tested. This stage is especially necessary to check the correct implementation of interfaces by several components.

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

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

### 3.2 Syntax of the Considered Erlang Subset

The described generator and interpreter for flow graphs do not work on the whole set of Erlang constructs. We rather restrict ourselves to the subset defined in Fig. 1, that essentially covers the sequential part of Erlang without some syntactic sugar. Definitions consisting of a $\star$ are not needed here, and are therefore omitted. Infix operators are considered as ordinary functions. Due to the importance of the BIF (built in function) *throw* for the control flow, it has the state of a syntactic keyword in this work. In the following when speaking of a first order function call we mean a call of the form $fn(e_1, e_2, \ldots, e_k)$ with a function name $fn$, and a higher order function call has the form $e_0(e_1, e_2, \ldots, e_k)$.

## 4 Structure of the Test Analyzer

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

1. Flow graph generation.
2. Interpretation of one or several tests in the flow graph.
3. Evaluation of the tests according to a specified coverage criterion.

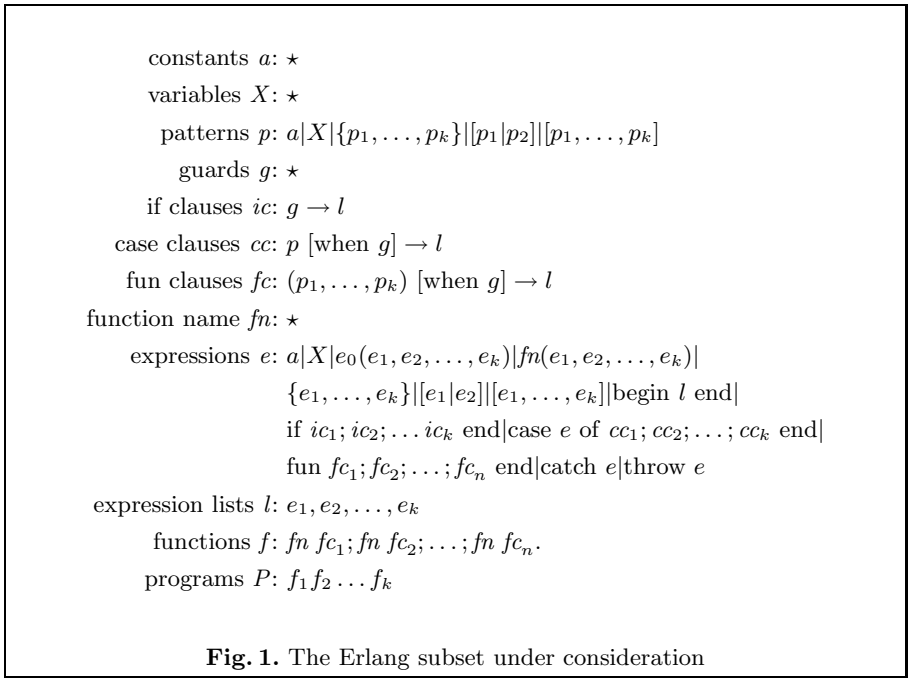


**Fig. 1.** The Erlang subset under consideration

### 4.1 Flow Graph Generation

The generation of flow graphs is described in other publications [14, 15] and therefore just sketched here for completeness. Given an Erlang module or a list of modules, the flow graph generation consists of the following four steps.

1. The modules are read and preprocessed to meet the following properties.
   - Every function has unique entry and exit points.
   - Each function has an *import node* indicating the local definition of the formal parameters, a *context node* indicating the local definition of the variables taken from the context of the function definition, and a *return node* denoting the leaving of the functions code and the returning of the value bound to the return variable.
   - All arguments to operations (function calls, structure generation, ...) are variables.
   - The return variable containing the return value of the function is bound before reaching the return node.
2. Flow graphs for the individual functions are generated. For each expression a node is generated (especially containing a unique node number within the module), and the nodes are connected in a straightforward manner. There exists a number of different node representations for different expressions in the source code [15].

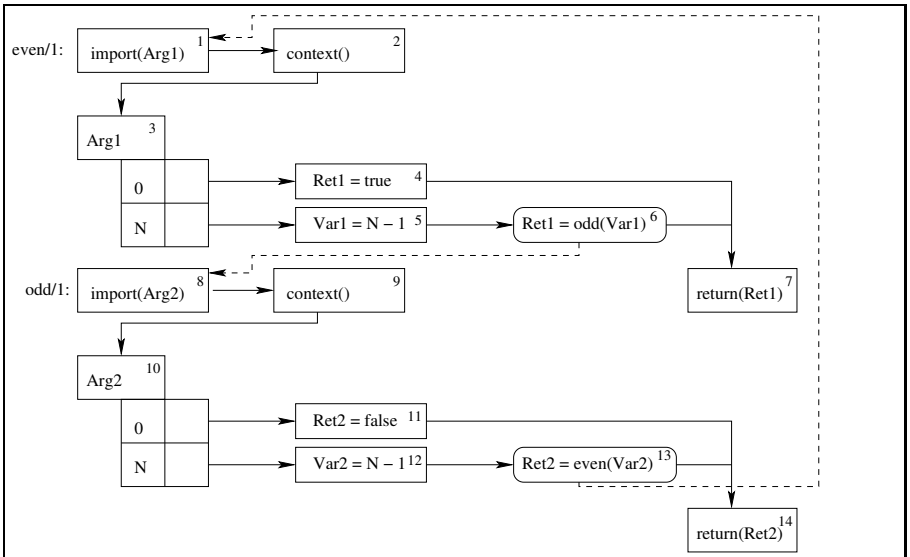


**Fig. 2.** Flow graph of the even-odd example

3. For each function call in the graph the possible destinations of the call are calculated and the corresponding call edges are introduced. For first order calls the destination is given by the called function; for higher order calls, an iterated data flow analysis process is necessary to compute all possible destinations. Call edges represent the control flow during calling a function *and* the return from the call [15].
4. For each `throw` expression in the code, indicating a non-local return, the corresponding destinations given by `catch` expressions are calculated and edges are introduced.

*Example 1.* 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.

### 4.2 The Graph Interpreter

A specialized interpreter for evaluating test cases is necessary to generate the input for the following step of evaluating the test cases. Besides the result of a test, the interpreter provides the trace through the flow graph during the test evaluation.

*Example 2.* Consider the flow graph from Ex. 1 and the test `even(1)`. This test yields the result `false` and the trace

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

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

Details on the design of the interpreter and a discussion of its properties are given in the following sections.

### 4.3 Judging the Test Coverage

Given the flow graph and the execution trace from the interpretation of some test cases, we can calculate the parts of the flow graph that were already covered by the tests. Several coverage criteria are known for imperative programming languages [16], some of which are expected to be usable for functional programming as well [15]. We just want to point out here, that the time consumption for checking the coverage rate of a given test set can heavily differ according to the chosen coverage criterion. Even quite simple coverage criteria like the node coverage criterion, or, in case of a complex control flow with non-local returns and higher order functions, the edge coverage criterion, can, however, be quite helpful in testing.

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

## 5 Design of the Graph Interpreter

In order to judge a given test set for coverage, the evaluation of the tests must be performed in a supervised manner. The goal of supervising a test is the computation of the trace the test takes through the flow graph. A collection of such traces can be analyzed according to the different coverage criteria afterwards. Precisely, the supervised execution is just necessary for the program parts (usually one or several modules) under test which are called the *supervised modules* in the following.

Two ways of supervised processing are possible: an interpreter for flow graphs or a modified compiler/runtime system for Erlang.

Since we are interested in easy to maintain code with a clear semantics for the prototype of our test evaluation system, we prefer an interpreter for the flow graph. In contrast to the modified compiler or runtime system, the implementation of an interpreter is self contained without a need for modifying existing code. Furthermore, a testing tool based on an interpreter only needs the flow graph as representation for the modules under supervision, while the alternative approach also needs a specialized version of the compiled code.

The main disadvantage of a simple interpreter as presented here is a loss in runtime performance and the loss of the space optimization for tail recursive calls. To answer the question whether these restrictions are tolerable, Sec. 6 contains several measurements and a discussion of their results.

The main function of the interpreter expects the following inputs.

- A structure (usually given as a node from the flow graph).
- A context mapping bound variables to their values.
- The flow graph (needed for performing function calls). For convenience reasons the current module in the graph is provided as an extra argument.

Given these arguments, the interpreter performs a case distinction on the form of the given structure. Simple values (like numbers and characters) are returned; for composed values the arguments are calculated and composed, and the values for variables are looked up in the context.

In evaluating a function call the interpreter has to distinguish the following cases for the called function.

- The called function is given by its module name and function name, and the corresponding module is a supervised one. In this case the body of the function in the flow graph is interpreted, using an updated context.
- The called function is given by its module name and function name, but the corresponding module is not a supervised one. The call is passed to the runtime system for evaluation. (The same holds for the predefined operators known in Erlang.)[1]
- The called function is given by an Erlang value. In order to be successful, the value must denote a fun.[2] The call to this fun is evaluated by the runtime system.
  Whenever a fun-value is generated by the interpreter, the execution of the fun has to be done by the interpreter as well. Therefore a fun is generated that just calls the interpreter with the appropriate context and the body of the intended fun. This ensures that the fun is always executed by the interpreter, even when called from outside the supervision (e.g. after the fun has been passed to a not supervised library function as argument).

The mechanism of non-local returns by *catch* and *throw* is implemented by using this mechanism in Erlang directly. A *catch* expression is evaluated by wrapping a catch around the call of interpreter for the argument. A *throw* just throws its argument directly. This approach has the disadvantage that non-local returns cannot be used by the interpreter (at least without great care over large distance), but since there might occur both, catches and throws within code evaluated by the runtime system, this restriction cannot be avoided anyway.

[1] Functions just given by their name (denoting calls within the same module) cannot occur here any more, because they are extended by the correct module name during preprocessing.

[2] Lambda closures are called *funs* in Erlang and are denoted by the keyword *fun*.

The main function of the interpreter returns a pair consiting of the evaluation result, and a context that might be needed for further sub-evaluations. The collection of execution traces makes use of the Erlang process concept. Before evaluating a node, the interpreter sends the node number (and in case of a module switch also the module name) to a process that collects the trace and provides it on request. This is necessary to return the trace of program parts that were called from outside the supervision.

## 6 Properties of the Graph Interpreter

The graph interpreter described in the section before is designed to offer a straightforward semantics and easy changes for experiments. In this section the practical use of the prototype is discussed, by measuring the maximal recursion depth and the runtime for several small test programs.

All tests were performed on an unloaded Sun Ultra SPARC 60 with a 450 MHz Microprocessor and 1,25 GB of memory.

### 6.1 The Test Programs

We start the discussion of the tests by describing the programs that were used for the tests.

The following module `loop` contains the base loop that is used by all the test programs.

```
-module(loop).
-export([l/1]).

l(0) -> 0;
l(X) -> l(X - 1).
```

The loop program is extended to perform an addition operation in each iteration yielding the module `add`.

```
-module(add).
-export([l/1]).

l(0) -> 0;
l(X) -> 1 + l(X - 1).
```

An alternative operation is the generation of lists that is performed by the module `cons`.

```
-module(cons).
-export([l/1]).

l(0) -> [];
l(X) -> [X | l(X-1)].
```

The following module `fun_gen` generates a fun in every iteration.

```
-module(fun_gen).
-export([l/1]).

l(0) -> 0;
l(X) ->
    F = fun l/1,
    l(X-1).
```

The call to funs is tested by the module `fun_call` that wraps the loop by a fun.

```
-module(fun_call).
-export([l/1]).

l(A) ->
    F = fun(0, _) -> 0;
           (X, G) -> G(X-1, G)
        end, % fun
    F(A, F).
```

Due to the second parameter that has to be carried through by `fun_call`, the structure is no longer directly comparable to `loop`. We therefore have another module `loop_call` that represents the structural differences of `fun_call` and `loop`.

```
-module(loop_call).
-export([l/1]).

l(A) ->
    F = fun(0, _) -> 0;
           (X, G) -> G(X-1, G)
        end, % fun (is never called, but just passed around)
    t(A, F).

t(0, _) -> 0;
t(X, G) -> t(X-1, G).
```

### 6.2 Thresholds for the Recursion Depths

As already mentioned, our interpreter is not able to perform space optimizations on tail recursive calls. Since Erlang relies on the fact that tail recursive calls can be evaluated in constant space, the interpreter will show unintended behaviour for large recursion depths.

Table 1 shows for each of the test programs the largest argument that does not break the interpreter because of a heap overflow.

| Program | Max. argument |
|---|---|
| `loop` | 2726568 |
| `add` | 1871991 |
| `cons` | 2507590 |
| `fun_gen` | 1523222 |
| `fun_call` | 1728769 |
| `loop_call` | 2010925 |

**Table 1.** Maximum arguments to the individual functions

All test programs allow a recursion depth of over 1.5 million. As one could expect, the highest recursion depth is found for `loop` that spends the least heap memory for calculations within each recursion step.

### 6.3 Runtime Measurements

For the runtime measurements each value was generated as the average of 8 individual measurements. Runtimes are given in milliseconds as reported by the Erlang profiler `fprof` [1]. The measurements just take into account the compilation into the portable *beam* code, but not the HiPE compiler [17] that supports compilation into native machine code on several platforms.

As a basis of the analysis we consider the individual programs for the arguments 1, 10, 100, 1000, and 10000 when evaluated in the standard runtime system. The results are presented in Fig. 3.

Besides a little overhead for small arguments, we get a linear correspondence between the argument and the runtime. The runtimes of the individual programs for the same argument are quite similar; the difference is bounded by a factor of 1.7 in the tests.

The same overview for the evaluation of the tests in our interpreter is presented in Fig. 4

Again we have a linear correspondence between the argument and the runtime, with an overhead for small argument sizes, that has, however, a smaller impact than in the case of standard evaluation. Again the runtimes of the individual programs for the same argument are quite comparable with a maximum difference factor of 2.0.

While the most expensive versions in standard evaluation were the value manipulations by `add` and `cons`, the function calls and the higher number of arguments in `loop_call` and `fun_call` have a stronger influence on the interpreted runtimes.

Given a test case, the most interesting question is, how much more time the interpreter needs to evaluate it compared to the standard runtime system. Figure 5 presents the factors of the interpreted runtime divided by the standard runtime.

Since the standard evaluation showed a higher overhead for small arguments than the interpreted evaluation, the factors are quite small for the argument 1 (between 20 and 30). They increase up to 120 to 250 for the argument 10000, and

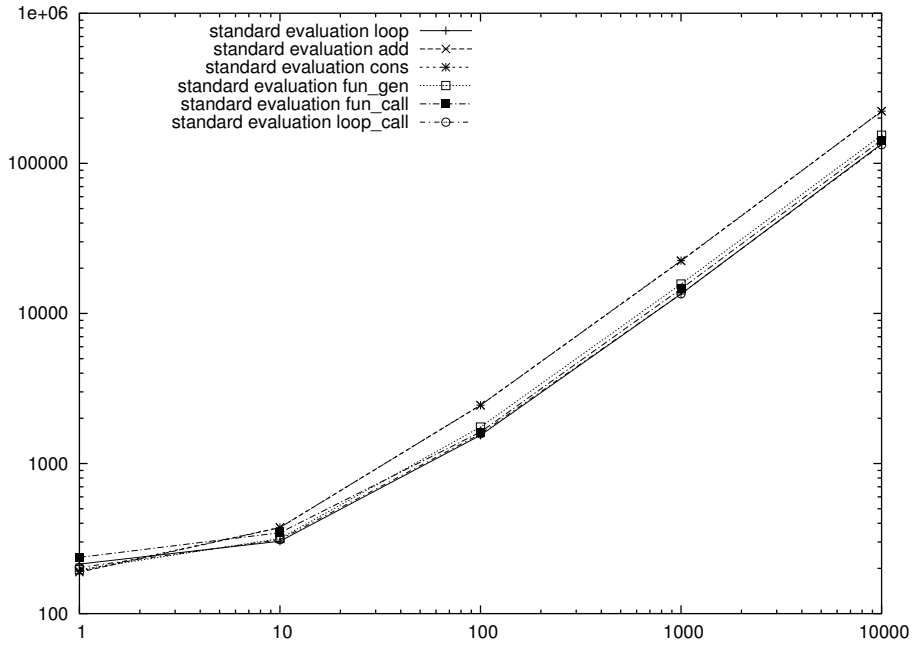


**Fig. 3.** Runtimes for standard evaluation

from the form of the graphs we can expect that the difference factor converges to values somewhere near the measurements for the argument 10000.

It turns out that the higher number of arguments in `loop_call` and `fun_call` causes the highest difference between interpreted and standard runtime for these two examples.

### 6.4 Discussion of the Results

Putting the above results together, the described interpreter allows for a recursion depth that is sufficient for testing program fragments of several hundreds to thousands of lines in most cases. Especially for small tests, the runtime ratio between standard and interpreted evaluation is not to bad because of the overhead of the standard system for small tests.

For large test sets we expect to see a mix of the scenarios simulated by the analyzed programs. In every case we expect the ratio between standard evaluation and interpretation not to exceed 500, even for functions with high arity and operations that are expensive in the interpretation.

Though the increase of time consumption by the flow graph interpreter compared to the standard evaluation cannot be ignored, we expect it to be acceptable in practical use for component testing because of the following reasons.

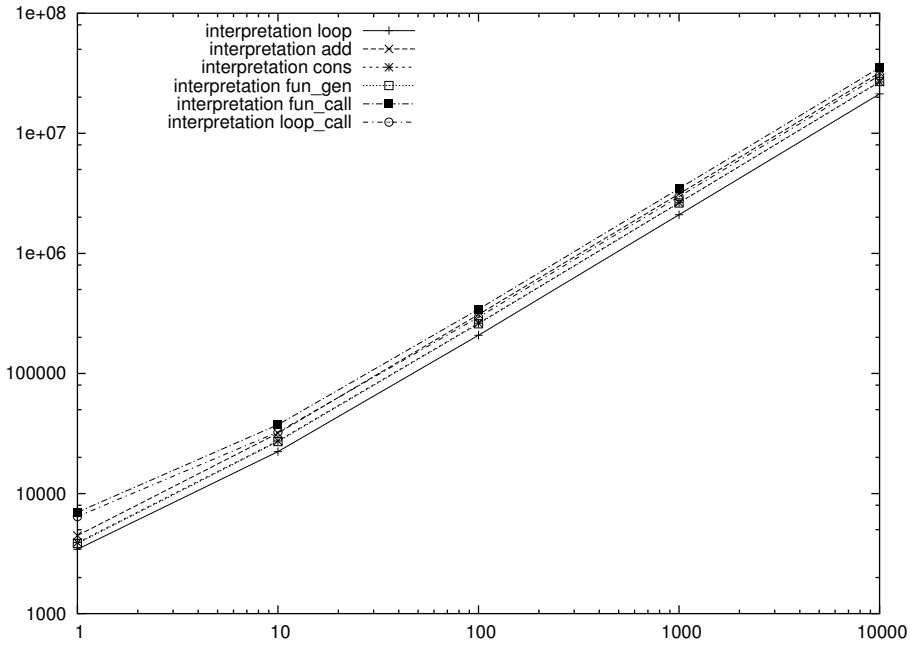


**Fig. 4.** Runtimes for interpretation

- The program fragments put to test during component testing are small. They should not exceed a single module without a good reason.[3]
- The test cases are usually small and simple. This is because unnecessarily complex test cases make it harder to track down a misbehavior of a program to a corresponding programming error.
- Much component testing can be done in batch mode where the runtime is less critical than in interactive mode. Test sets are often designed according to further criteria besides coverage, so that the design is completed before performing a coverage test, and only few test cases are added interactively in order to reach coverage. (This is especially the case for regression testing where the test set from a previous program version is already available.)
- In situations where the runtime is a real matter, many tests can be performed without using the interpreter. The coverage analysis can be done when the performed tests do not show any misbehavior of the tested code any more.

## 7 Conclusions and Future Work

The supervised evaluation of test cases has been shown important in judging the appropriateness of a test set for a program fragment according to a certain

[3] Note that testing the software in larger blocks is deferred to the integration testing and system testing stages.

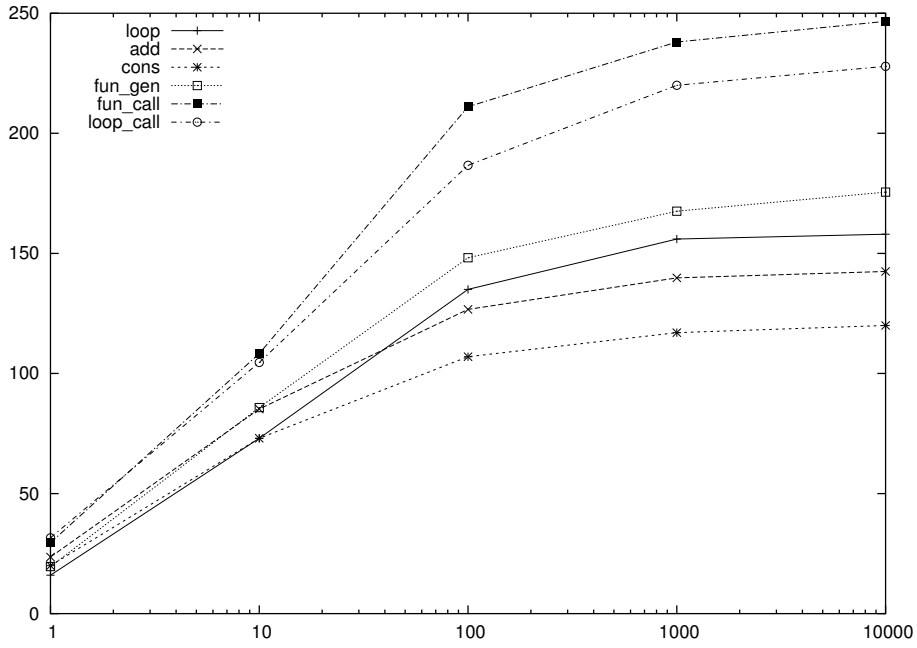


**Fig. 5.** Difference factors between interpreted and standard execution

coverage criterion. The solution described here is an interpreter that evaluates the tests in the flow graph and tries to represent the semantics of the language as seen by the programmer as precisely as possible.

Since source code directed testing usually applies to relatively small software components, the interpreter is just used for the supervised modules (i.e. the set of modules that is currently under testing). Calls to unsupervised modules are passed to the runtime system. Because of this, the interpretation of a fun generation must indeed return a fun, that calls the interpreter with the appropriate context and function body whenever evaluated. This is necessary, because supervised funs can be called outside the supervision, e.g. when passed to a library function as argument. The interaction between the interpreted parts of the program and those evaluated by the runtime system also enforces the catch-throw mechanism of Erlang to be used in the interpreter to implement the catch-throw functionality.

Measurements showed a maximal possible recursion depth, that exceeded 1.500.000 for all tests on the used machine and should therefore be more than sufficient for practical use, even though the interpreter does not optimize tail recursion.

The delay by a factor between 20 and 250 compared to the execution by the Erlang runtime system is expected to be tolerable in the intended application scenario, especially allowing for batch execution of a larger number of tests in the

most cases. Less runtime consuming implementations are possible here (e.g. the compilation of the tested modules to an enriched beam code), but at the cost of a more complex implementation and a less clear and less extendable semantics.

Future work on our test system include the implementation and evaluation of several coverage criteria, and the evaluation of the source code directed testing approach for Erlang in a larger case study.

## References


1. NN: (Tools version 2.3) Documentation for Erlang OTP R9C.
2. van den Berg, K.: Software Measurement and Functional Programming. (1995)
3. Shivers, O.: Control-flow analysis in Scheme. In: Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. (1988) 164–174
4. Ashley, J.M., Dybvig, R.K.: A practical and flexible flow analysis for higher-order languages. ACM Transactions on Programming Languages and Systems **20** (1998) 845–868
5. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP'00). Volume 35.9 of ACM Sigplan Notices., N.Y., ACM Press (2000) 268–279
6. Rothermel, G., Burnett, M., Li, L., Dupuis, C., Sheretov, A.: A methodology for testing spreadsheets. ACM Transactions on Software Engineering and Methodology **10** (2001) 110–147
7. Rothermel, G., Li, L., DuPuis, C., Burnett, M.: What you see is what you test: A methodology for testing form-based visual programs. In: Proceedings of the 1998 International Conference on Software Engineering, IEEE Computer Society Press/ACM Press (1998) 198–207
8. Rothermel, K.J., Cook, C.R., Burnett, M.M., Schonfeld, J., Green, T.R.G., Rothermel, G.: WYSIWYT testing in the spreadsheet paradigm. In: Proceedings of the 22nd International Conference on Software Engineering, ACM Press (2000) 230–239
9. Gill, A.: Debugging Haskell by observing intermediate data structures. In: Proceedings of the 4th Haskell Workshop. Technical report of the University of Nottingham. (2000)
10. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming **1997** (1997)
11. Chitil, O.: A semantics for tracing. In: Draft Proceedings of the 13th International Workshop on Implementation of Functional Languages, IFL. (2001)
12. Wallace, M., Chitil, O., Brehm, T., Runciman, C.: Multiple-view tracing for Haskell: a new hat. In: Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop, Firenze, Italy. (2001) 151–170
13. Liggesmeyer, P.: Software-Qualität: Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag, Heidelberg, Berlin (2002)
14. Widera, M.: Towards flow graph directed testing of functional programs. In Trinder, P., Michaelson, G., eds.: Draft Proceedings of the 15th International Workshop on the Implementation of Functional Languages. (2003)
15. Widera, M.: Flow graphs for testing sequential erlang programs. In: Proceedings of the 3rd ACM SIGPLAN Erlang Workshop. (2004) to appear.

16. Zhu, H., Hall, P., May, J.: Software unit test coverage and adequacy. ACM Computing Surveys **29** (1997) 366–427
17. Johansson, E., Pettersson, M., Sagonas, K.F.: A high performance erlang system. In: Principles and Practice of Declarative Programming. (2000) 32–43