Change Impact Analysis [1]

Melinda Tóth, István Bozó and Zoltán Horváth

Eötvös Loránd University
Budapest, Hungary

November 3, 2011
Erlang User Conference, Stockholm

[1] Supported by TECH_08_A2-SZOMIN08, KMOP-1.1.2-08/1-2008-0002 and European Regional Development Fund (ERDF)

Outline

1. Motivation
2. Background
3. Intermediate Source Code Representation
4. Test Case Selection

Motivation

- Refactoring legacy source code – **RefactoringErlang**

Motivation

- Refactoring legacy source code – **RefactoringErlang**
- Regression test becomes necessary

Motivation

- Refactoring legacy source code – **RefactoringErlang**
- Regression test becomes necessary
- Reduce the number of test suits

Motivation

- Refactoring legacy source code – **RefactoringErlang**
- Regression test becomes necessary
- Reduce the number of test suits
- Perform *Program Slicing* based Change Impact Analysis

Refactoring – Source Code Transformation

- Syntactic source code transformation
- Preserves the meaning
- Pre-conditions
- Static program analysis

RefactorErl

- Static source code analyser and transformer tool
- Platform for source code transformations – 24 implemented refactorings
    - Rename
    - Move definition
    - Expression structure
    - Function interface
    - Parallelisation
- Structural source code analysis – Clustering
- Support for program comprehension
    - Call graph visualisation
    - Dependency analysis
    - Semantic Query Language / Metric Query Language
- Several interfaces

Web Interface

Applications Places System

RefactorErl Queries

localhost:8001/main

Nov 2 19:56

RefactorErl

Queries Files Errors Dependency Graphs Log out melinda

mods[name=appmon_txt].funs

Save as skeleton

Run

Previous Queries | Running Queries | File Browser | Skeletons

Previous Queries

My queries

- mods.funs
- mods[name=appmon].funs
- mods[name=appmon_txt].funs

mods[name=appmon_txt].funs

appmon_txt.erl

- appmon_txt:print/1
- appmon_txt:start/0
- appmon_txt:start/1
- appmon_txt:fprint/1
- appmon_txt:init/1
- appmon_txt:handle_call/3
- appmon_txt:handle_cast/2
- appmon_txt:handle_info/2
- appmon_txt:terminate/2
- appmon_txt:setup_base_win/0
- appmon_txt:default_status/0
- appmon_txt:setup_opts/1
- appmon_txt:do_insert_text/1
- appmon_txt:scroll_to_last_line/0
- appmon_txt:do_load_file/1
- appmon_txt:ui_load/0
- appmon_txt:resize/2
- appmon_txt:print_status/1
- appmon_txt:setup_opt/1
- appmon_txt:do_lock/0
- appmon_txt:i_load_file/1
- appmon_txt:i_do_clear/0
- appmon_txt:get_file_list/0
- appmon_txt:ui_list_dialog/3
- appmon_txt:editor/0
- appmon_txt:label_h/0

```
%%
%% %CopyrightBegin%
%%
%% Copyright Ericsson AB 1996-2009. All Rights Reserved.
%%
%% The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved online at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitati
%% under the License.
%%
%% %CopyrightEnd%
%%-------------------------------------------------------------
%%
%% Simple text viewer
%%
%%-------------------------------------------------------------

-module(appmon_txt).
-export([start/0, start/1, print/1, fprint/1]).

%% gen_server stuff
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
```

Inbox - toth_m@i... | slides | euc11.pdf - Adob... | [emacs23@melin... | [melinda@melind... | [melinda@melind... | RefactorErl Queri...

Change Impact Analysis

- Calculate the impact of a change on the source code
- Changes made by manually or by a refactoring tool
- Identify the affected code parts
- Base on static program analysis and Program Slicing techniques
- Assistance in testing and debugging

Brief Overview of Program Slicing

- "A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest (slicing criterion)." (Weiser, 1979)
- Many forms of slicing:
  - Static – Dynamic
  - Forward – Backward
  - Executable – Not executable
  - Intraprocedural – Interprocedural
  - Dependence Graph – Data-flow equations – Information-flow relations based approaches

Our Approach

- Slicing in a graph representation of the source code
- Different levels of intermediate source code representation
- Steps in building the Dependency Graph:
  1. **Control-Flow Graph**
  2. Postdominator Tree
  3. **Control Dependence Graph**
  4. **Data-Flow Graph**
  5. **Behavior Dependence Graph**

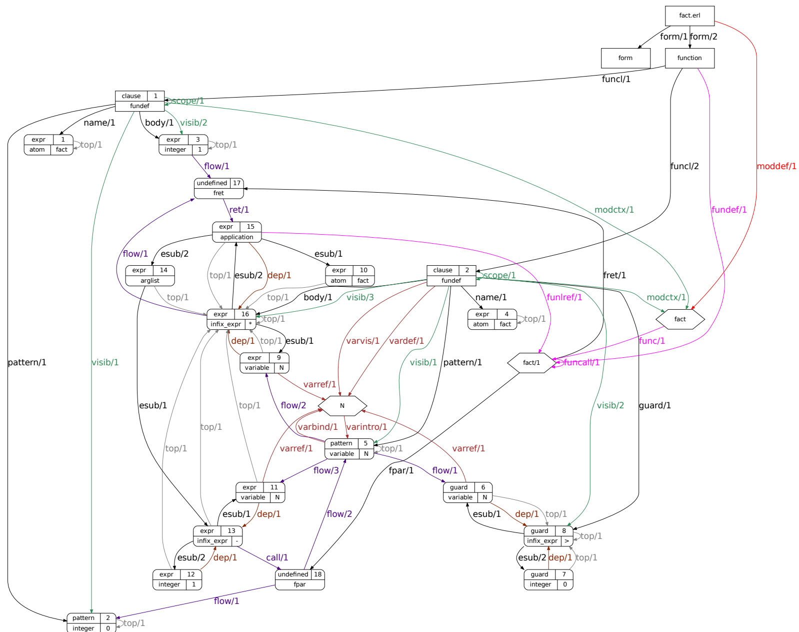
Semantic Program Graph of RefactorErl

1. Lexical level
   - Tokens
   - Preprocessing
   - Comments, whitespace
2. Syntactic level
   - Abstract Syntax Tree
3. Semantic level
   - Module, function, record, variable nodes
   - Links to definition and reference points
   - Identifies side-effects, dynamic function references, etc.

Factorial function

```
-module(fact).

fact(0) -> 1;
fact(N) when N>0 ->
    N*fact(N-1).
```


fact.erl
form/1
form/2
form
function
funcl/1
funcl/2
moddef/1
fundef/1
clause 1
fundef
scope/1
name/1
body/1
visib/2
expr 1
atom fact
top/1
expr 3
integer 1
top/1
flow/1
undefined 17
fret
ret/1
modctx/1
expr 15
application
flow/1
esub/2
esub/1
top/1
esub/2
dep/1
expr 14
arglist
expr 10
atom fact
clause 2
fundef
scope/1
fret/1
top/1
top/1
body/1
visib/3
name/1
funlref/1
modctx/1
expr 16
infix_expr *
top/1
expr 4
atom fact
top/1
fact
dep/1
top/1
esub/1
varvis/1
vardef/1
func/1
pattern/1
visib/1
expr 9
variable N
visib/1
pattern/1
fact/1
funcall/1
varref/1
esub/1
top/1
flow/2
N
visib/2
guard/1
top/1
varbind/1
varintro/1
pattern 5
variable N
top/1
top/1
varref/1
varref/1
flow/3
fpar/1
flow/1
expr 11
variable N
guard 6
variable N
esub/1
dep/1
flow/2
esub/1
dep/1
top/1
expr 13
infix_expr -
guard 8
infix_expr >
top/1
esub/2
dep/1
call/1
esub/2
dep/1
top/1
expr 12
integer 1
undefined 18
fpar
guard 7
integer 0
flow/1
pattern 2
integer 0
top/1

Data-Flow Analysis

- Technique for gathering information about the possible set of values calculated at various points in a program
- Goal: detecting direct and indirect data-flow and data-dependency among expressions
- Data Flow Graph - DFG
- The nodes of the DFG are the expressions
- An edge represents direct data-flow relation between two nodes
- Data-flow reaching to calculate indirect flow

Simple Data-Flow Rule for Erlang

p is a binding of a variable:
n is an occurrence of the same variable

$p \xrightarrow{f} n$

```
A = 3,
...
B = A + 2
```

$\$3 \xrightarrow{f} \$A$

$\$A \xrightarrow{f} \$A$

Function Call Data-Flow Rule

e_0:
$f(e_1, \ldots, e_n)$

f/n:
$f(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow e_1^1, \ldots, e_{l_1}^1$;
$\vdots$
$f(p_1^m, \ldots, p_n^m)$ when $g_n \rightarrow e_1^n, \ldots, e_{l_n}^n$

$$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$$

$$e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$$
$$\vdots$$
$$e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$$

Function Call Data-Flow Rule

e_0:
$f(e_1, \ldots, e_n)$

f/n:
$f(p_1^1, \ldots, p_n^1)$ when $g_1 \rightarrow e_1^1, \ldots, e_{l_1}^1$;
$\vdots$
$f(p_1^m, \ldots, p_n^m)$ when $g_n \rightarrow e_1^n, \ldots, e_{l_n}^n$

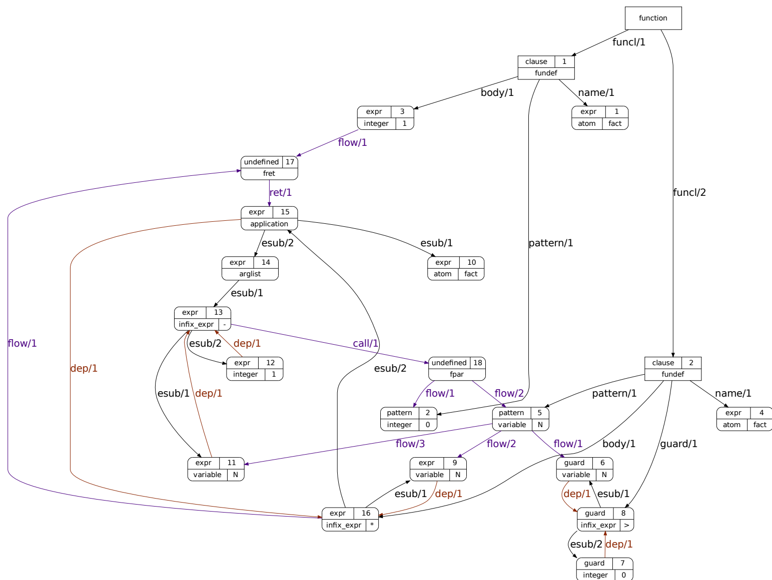
$e_{l_1}^1 \xrightarrow{f} e_0, \ldots, e_{l_m}^m \xrightarrow{f} e_0$
$e_1 \xrightarrow{f} p_1^1, \ldots, e_1 \xrightarrow{f} p_1^m$
$\vdots$
$e_n \xrightarrow{f} p_n^1, \ldots, e_n \xrightarrow{f} p_n^m$

```
myfun(A, B)->
    C = A + B,
    {A, B, C}.

mycall()->
    {E1, E2, E3} = myfun(4,2),
    E1.
```

$\$\{A, B, C\} \xrightarrow{f} \$myfun(4, 2)$

$\$4 \xrightarrow{f} \$A, \$2 \xrightarrow{f} \$B$


function
funcl/1
funcl/2
clause 1
fundef
body/1
name/1
pattern/1
expr 3
integer 1
expr 1
atom fact
flow/1
undefined 17
fret
ret/1
expr 15
application
esub/2
esub/1
expr 14
arglist
expr 10
atom fact
esub/1
expr 13
infix_expr -
esub/2
dep/1
call/1
expr 12
integer 1
undefined 18
fpar
flow/1
flow/2
esub/2
esub/1
dep/1
dep/1
flow/1
clause 2
fundef
pattern/1
name/1
pattern 2
integer 0
pattern 5
variable N
expr 4
atom fact
flow/3
flow/2
flow/1
body/1
guard/1
expr 11
variable N
expr 9
variable N
guard 6
variable N
esub/1
dep/1
dep/1
esub/1
expr 16
infix_expr *
guard 8
infix_expr >
esub/2
dep/1
guard 7
integer 0

Data-flow reaching

$$n \overset{0f}{\rightsquigarrow} n \qquad \text{(reflexive)}$$

$$\frac{n_1 \overset{f}{\rightarrow} n_2}{n_1 \overset{0f}{\rightsquigarrow} n_2} \qquad \text{(f-rule)}$$

$$\frac{n_1 \overset{c_i}{\rightarrow} n_2, \quad n_2 \overset{0f}{\rightsquigarrow} n_3, \quad n_3 \overset{s_i}{\rightarrow} n_4}{n_1 \overset{0f}{\rightsquigarrow} n_4} \qquad \text{(c-s-rule)}$$

$$\frac{n_1 \overset{0f}{\rightsquigarrow} n_2, \quad n_2 \overset{0f}{\rightsquigarrow} n_3}{n_1 \overset{0f}{\rightsquigarrow} n_3} \qquad \text{(transitive)}$$

Message sending rule

e_0:
$e_1 \; ! \; e_2$
e':
receive
p_1 when $g_1 \rightarrow$
$e^1_1, \ldots, e^1_{l_1}$;
$\vdots$
p_n when $g_n \rightarrow$
$e^n_1, \ldots, e^n_{l_n}$
after
$e \rightarrow e_1, \ldots, e_s$
end

$$e_2 \xrightarrow{f} e_0$$

$$\mathbf{e_2 \xrightarrow{f} p_1, \ldots, e_2 \xrightarrow{f} p_n}$$

$$e^1_{l_1} \xrightarrow{f} e', \ldots, e^n_{l_n} \xrightarrow{f} e'$$
$$e_s \xrightarrow{f} e'$$

Detecting Message Passing via Data-Flow

$$\mathbf{e_2} \xrightarrow{\mathbf{f}} \mathbf{p_1}, \ldots, \mathbf{e_2} \xrightarrow{\mathbf{f}} \mathbf{p_n}$$

Detecting Message Passing via Data-Flow

- $spawn*(Mod, Fun, Args) \overset{0f}{\rightsquigarrow} e_1$ – backward data-flow

$$\mathbf{e_2 \xrightarrow{f} p_1, \ldots, e_2 \xrightarrow{f} p_n}$$

Detecting Message Passing via Data-Flow

- $spawn * (Mod, Fun, Args) \overset{0f}{\leadsto} e_1$ – backward data-flow
- $m \overset{0f}{\leadsto} Mod$ – backward data-flow
- $f \overset{0f}{\leadsto} Fun$ – backward data-flow

$$\mathbf{e_2 \xrightarrow{f} p_1, \ldots, e_2 \xrightarrow{f} p_n}$$

Detecting Message Passing via Data-Flow

- $spawn * (Mod, Fun, Args) \overset{0f}{\rightsquigarrow} e_1$ – backward data-flow
- $m \overset{0f}{\rightsquigarrow} Mod$ – backward data-flow
- $f \overset{0f}{\rightsquigarrow} Fun$ – backward data-flow
- $[Elem_1, ..., Elem_n] \overset{0f}{\rightsquigarrow} Args$ – backward data-flow

$$\mathbf{e_2} \overset{\mathbf{f}}{\rightarrow} \mathbf{p_1}, \ldots, \mathbf{e_2} \overset{\mathbf{f}}{\rightarrow} \mathbf{p_n}$$

Detecting Message Passing via Data-Flow

- $spawn * (Mod, Fun, Args) \overset{0f}{\rightsquigarrow} e_1$ – backward data-flow
- $m \overset{0f}{\rightsquigarrow} Mod$ – backward data-flow
- $f \overset{0f}{\rightsquigarrow} Fun$ – backward data-flow
- $[Elem_1, ..., Elem_n] \overset{0f}{\rightsquigarrow} Args$ – backward data-flow
- e' is $reachable$ $from$ $m : f/n$ – call-chain

$$\mathbf{e_2 \xrightarrow{f} p_1, \ldots, e_2 \xrightarrow{f} p_n}$$

Detecting Message Passing via Data-Flow

- $spawn * (Mod, Fun, Args) \overset{0f}{\leadsto} e_1$ – backward data-flow
- $m \overset{0f}{\leadsto} Mod$ – backward data-flow
- $f \overset{0f}{\leadsto} Fun$ – backward data-flow
- $[Elem_1, ..., Elem_n] \overset{0f}{\leadsto} Args$ – backward data-flow
- e' is $reachable$ $from$ $m : f/n$ – call-chain
- registered names – 2 bw data-flow, 1 fw data-flow + reg. name usage

$$\mathbf{e_2 \overset{f}{\rightarrow} p_1, \ldots, e_2 \overset{f}{\rightarrow} p_n}$$

Control-flow Analysis

- Technique for determining the control flow of a program
- Control-Flow Graph - CFG
- Every execution path
- The nodes of the CFG are the expressions
- An edge represents direct control-flow relation between two nodes

Simple Control-Flow Rule

e_0:
$e_1 \circ e_2$

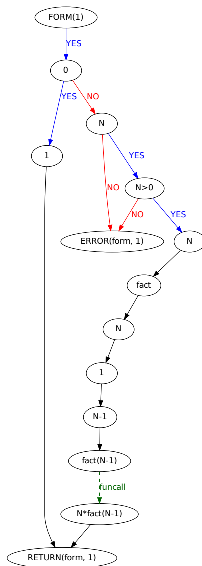
$e_0' \rightarrow e_1$
$e_1 \rightarrow e_2$
$e_2 \rightarrow e_0$

e_0:
$\{e_1, \ldots, e_n\}$

$e_0' \rightarrow e_1$
$e_1 \rightarrow e_2, \ \ldots, \ e_{n-1} \rightarrow e_n$
$e_n \rightarrow e_0$

Example CFG

```
fact(0) -> 1;
fact(N) when N>0 ->
    N*fact(N-1).
```



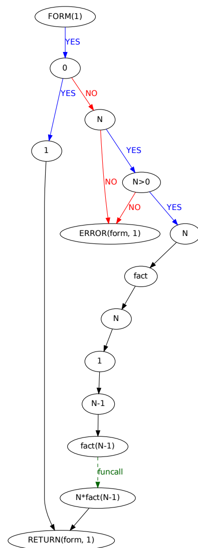
Control Dependency Analysis

- Control dependence is a relation when an expression of a program is evaluated if a previous expression evaluates in a way that allows its evaluation
- Eliminating unnecessary sequencing

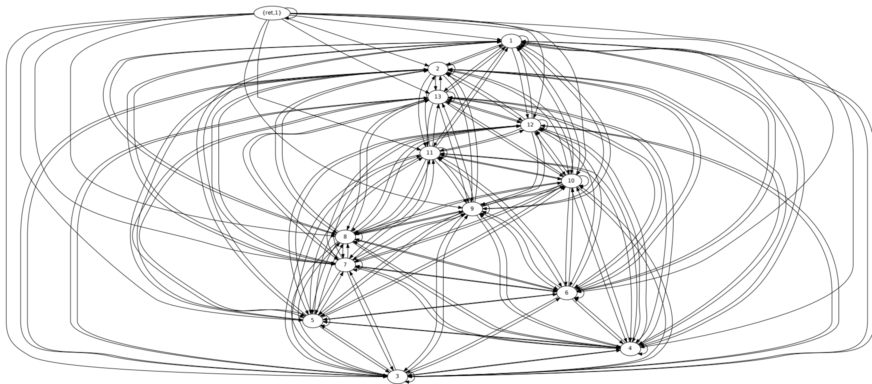
```
simplefun()->
    ...
    A = 2 + 4,
    B = 4 + 2,
    ...
```

Building Postdominator Tree

- Postdominator relation: **j** postdom **i**, if every execution path from **i** to **exit** includes **j**
- Immediate postdominator: **j** is ipostdom of **i**, if and only if **j** postdom **i** and does not exists a node **k** such that $\mathbf{i} \neq \mathbf{k}$ and $\mathbf{j} \neq \mathbf{k}$ for which **k** postdom **i** and **j** postdom **k**.

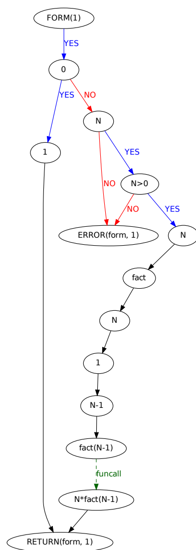


Building Postdominator Tree (cont.)

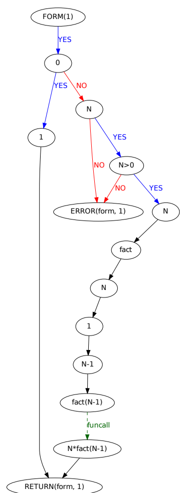


Building Postdominator Tree

- Postdominator relation: **j** postdom **i**, if every execution path from **i** to **exit** includes **j**
- Immediate postdominator: **j** is ipostdom of **i**, if and only if **j** postdom **i** and does not exists a node **k** such that **i** $\neq$ **k** and **j** $\neq$ **k** for which **k** postdom **i** and **j** postdom **k**.



Building Postdominator Tree (cont.)

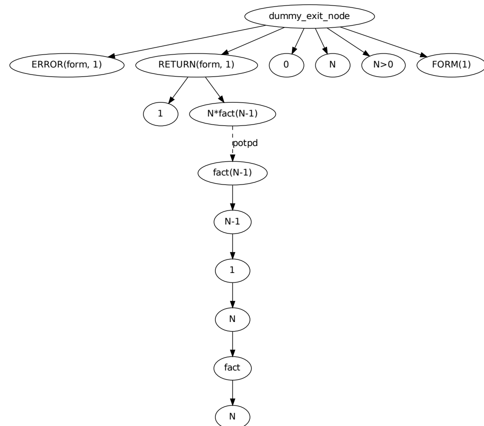


dummy_exit_node
ERROR(form, 1)
RETURN(form, 1)
0
N
N>0
FORM(1)
1
N*fact(N-1)
potpd
fact(N-1)
N-1
1
N
fact
N

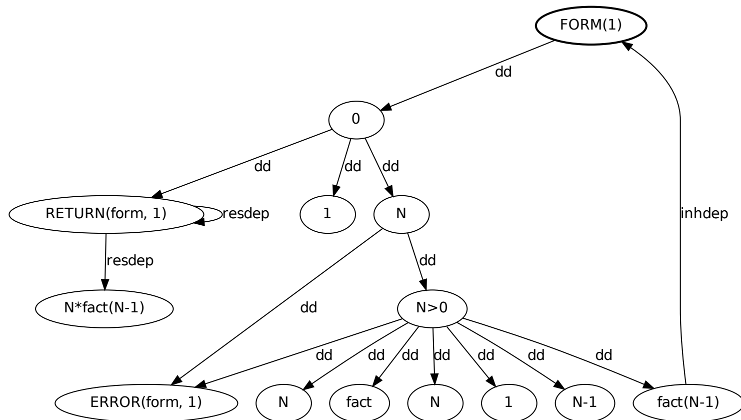
Building the Control Dependence Graph

j is control dependent from i iff

- Exists a path from i to j, and $\forall$ k from this path, $k \neq i \wedge k \neq j$: **j** postdominates **k**
- **j** does not postdominate **i**

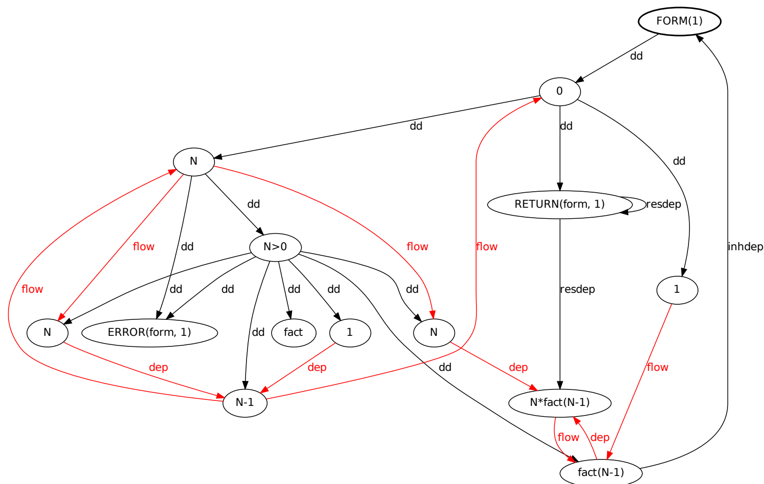


Building CDG (cont.)



Building a Dependence Graph

- Extend the control dependence graph with data dependency and data flow edges



Test Case Selection through Program Slicing

- Our goal: perform impact analysis through static program slicing
- Selecting test cases that are represented as Erlang functions (QuickCheck, EUnit, etc.)
- Not executable static forward slicing to select test suits of the program that should be rechecked due to a change at the selected point
- The slicing criterion is a set of vertices in the graph corresponding to the changed expressions in the source code after a refactoring
- Follow the control dependency and data dependency edges in the dependence graph (reachability problem in the dependence graph)

Summary

- Selecting test cases (QuickCheck, EUnit, etc.) after a change on the source code
- Slicing will be part of RefactorErl from January, 2012
- Components of the analysis toolset are already available:
  - Side effect analysis
  - Data-flow analysis to calculate values of an expression
  - Dynamic function call analysis
  - Detecting data-flow among processes
- Composing existing static analysis towards:
  - Deadlock detection
  - Detecting process structure based on data-flow relations
  - Detecting parallelisable program parts
  - Detecting design patterns

http://plc.inf.elte.hu/erlang