Impact Analysis of Erlang Programs Using Behaviour Dependency Graphs[*]

Melinda Tóth, István Bozó, Zoltán Horváth, László Lövei, Máté Tejfel, and Tamás Kozsik

Eötvös Loránd University, Budapest, Hungary
{toth_m,bozo_i,hz,lovei,matej,kto}@inf.elte.hu

Abstract. During the lifetime of a software product certain changes could be performed on its source code. After those changes a regression test should be performed, which is the most expensive part of the software development cycle. This paper focuses on programs written in a dynamic functional programming language Erlang, and discusses a mechanism that could select those test cases, which are affected by a change, i.e. altering the program on some point may have impact on the result/behaviour of those test cases. In the result of that analysis it is possible to reduce the number of necessary test cases, and after modifying the source code, just a subset of the test cases should be retested. The discussed approach introduces a behaviour dependency graph for Erlang programs to represent the dependencies in the source code. The impact of a change can be calculated by traversing the graph.

1 Introduction

Changes often happen in a software lifetime. These changes can be done manually by a programmer or using a refactoring tool. The phase "refactoring" [3] introduces a meaning preserving source code transformation, thus you change the structure of a program without altering its external behaviour. Refactoring could be done manually by a programmer or using a refactoring tool. The former case is tedious and error prone, the latter is safer and faster. A refactoring tool guarantees that the transformation does not change the meaning of the program and all the necessary changes will happen. However refactoring in Erlang [2] is not straightforward. The language is dynamically typed, so the syntactic and static semantic information sometimes may be not enough to guarantee a meaning preserving transformation, and the programmer want to test the behaviour of the transformed program. Since testing is a very expensive part of the software development process, we want to help the programmers to reduce the number of test cases which should be performed after a transformation or a sequence of transformations. Therefore we try to find the affected parts in the source code by analyzing the spread of the impact of performed changes. Assume we have found the affected code parts, then only a subset of test cases should be retested, those which are affected by the change.

[*] Supported by TECH_08_A2-SZOMIN08, ELTE IKKK, and Ericsson Hungary.

To find the affected parts we have to propagate the change of some data, therefore we introduce a behaviour dependency graph. During the generation of this graph we use static syntactic and semantic information based on the semantic program graph of the RefactorErl.

RefactorErl [6,5] is a refactoring tool for Erlang. To represent the source code the tool uses its own semantic program graph model, which contains lexical, syntactic and semantic information about the loaded Erlang programs. The graph is based on the AST. RefactorErl uses its own layout preserving parser to generate the syntax tree, then different semantic analyzers (function, variable, record, etc) extend the syntax tree to semantic program graph. The constructed program graph provides good interface for further source code analysis.

The rest of this paper is structured as follows. In Section 2 a motivating example is given. Section 3 introduces the used model of Erlang programs. Then the Section 4 introduces the behaviour dependency graph, the method of constructing the behaviour dependency edges and the way of retrieving dependency information from this graph. Section 5 presents related work, and Section 6 concludes the paper and discusses future work.

2 Motivating Example

For the sake of simplicity we demonstrate our mechanism in a more general example. We do not transform the source code by a refactoring and analyze its impact, rather we modify an element of a list (Figure 1 and 2) and then we estimate the impact of the data change to determine the test cases which should be retested.

Consider the following example (Figure 1), where we define the `tag_add/1` recursive function, which transforms the elements of the given list to a tagged tuple. We expect from this function, that it does not alter the length of the given list. The `test_tag_add/2` function is intended to describe this property of the `tag_add/2` function. The `len/1` function calculates and returns the length of the given list. In our example the `tag_list/0` function calls the `tag_add/2` function with a list containing two integers: `[1,2]`.

Assume, that the programmer modify the first element of the list `[1,2]` in the body of `tag_list/0`. This value flows into the variable `H1` (in `tag_add/2`) through the list construct. The result of the `tag_add/2` function is a list which spine does not depend on the value of `H1` variable, however the elements of this result list depend on the value of `H1`. Therefore, we have to detect whether the elements of the resulted list are used elsewhere in the code. It can be used those point in the program where the `tag_add/2` function is called. The `test_tag_add/2` function calls the `tag_add/2` function, thus it uses its return value, so the elements of the resulted list may be used. The `test_tag_add/2` function passes the result of the `tag_add(L1, Tag)` function call as an argument for the `len/1` function and calculates the length of the resulted list with `len/1`, but during the calculation `len/1` does not use the value of the elements coming form its argument (in the body of `len/1` the values of the elements

```
tag_add([], _Tag) ->
    [];
tag_add([H1|T1], Tag)->
    [{Tag, H1} | tag_add(T1, Tag)].

tag_list() ->
    tag_add([1,2], integer).

test_tag_add(L1, Tag)->
    len(L1) == len(tag_add(L1, Tag)).

len([]) ->
    0;
len([_H | T]) ->
    1 + len(T).
```

Fig. 1. The definition of `tag_add/2`

(`_H`) of the parameter list are not used). Therefore the result of the `len(L1)` and `len(tag_add(L1))` function calls do not depend on the values of the elements coming from the `L1` list and the return value of the `tag_add(L1)` function call. We can see, that changing any element of the list `[1,2]` under writing the program does not have any impact on the result of these function calls, thus does not have any impact on the return value and on the behaviour of the `test_tag_add/2` function, so we must not retest it.

However the elements of the list depend on the operation (a tuple constructor), in this case the structure of the list is independent of this operation. The change of one list element does not always has impact on the structure of the list or the context of the list usage.

In the second part of the example we give the definition of a similar function, but the behaviour of these functions are different. On Figure 2 we define the `tag_filter/2` recursive function, which filters the elements of the list with a given tag `Tag`. This function selects a sublist of the given list, it may throw out element from the list. Thus changing the elements of the parameter list may affect the spine of the resulted list. This function may keep or decrease the length of the list (depending on the given tag and the content of the given list), but it can never increase it. The `test_tag_filter/2` function is intended to describe this property of the `tag_filter/2` function. The `filter_list/0` function calls the `tag_filter/2` function to select the elements with key `integer` from the list `[{integer, 1}, {atom, a}]`.

Assume, that the programmer change the first element of the list in the function call `tag_filter([{integer, 1}, {atom, a}], integer))`. The impact of this modification flows into the `H2` variable, in the same way as in the first part of our example where the elements of the `[1,2]` list flows into the `H1` variable, but after this point the difference between the `tag_add/2` and `tag_filter/2` functions shows up. The return value of the `tag_filter/2` function depends on the

```
tag_filter([], _Tag) ->
    [];
tag_filter([H2|T2], Tag) ->
    case H2 of
        {Tag, _Elem} -> [H2 | tag_filter(T2, Tag)];
        _     -> tag_filter(T2, Tag)
    end.

filter_list() ->
    tag_filter([{integer, 1}, {atom, a}], integer).

test_tag_filter(L2, Tag) ->
    len(L2) >= len(tag_filter(L2, Tag)).

len([]) ->
    0;
len([_H | T]) ->
    1 + len(T).
```

Fig. 2. The definition of `tag_filter/2`

elements passed as its arguments, because the case-expression depends on the result of the `H2` expression, thus the return value of the `tag_filter/2`function depends on the elements of its parameter list (`H2` gets its value from that list). The expression `len(tag_filter(L2, Tag))` comprehends a function call of `tag_filter/2`, which return value depends on the elements of its parameter list, thus the result of the entire expression depends transitively on the elements of the argument list. Therefore, any change on the elements of the input list may have an impact on the `test_tag_filter/2` function.

To detect these dependencies in an Erlang program, we have to define a dependency graph for Erlang. It should contain the data flow edges and behaviour dependency edges, too. The rules when the value or the behaviour of an expression has an impact on an other expression (for example, the case expression in the mentioned example depends on the expression `H2`) are defined in the following sections.

3 A Partial Model for Erlang Programs

In Section 4 we use the Erlang syntax shown in Figure 3. This syntax is a subset of the Erlang syntax presented in [4]. The symbol P denote the patterns can be used in Erlang, E represents guard expressions and expressions that can be defined in the language, and F denotes the named functions.

The presented syntax contains some simplification:

– Guard expressions are represented as expressions with some restrictions. Guard expressions can contain "guard" built-in function calls or type tests. The infix guard expressions are arithmetic or boolean expressions, or term comparisons. Guards can contain only bound variables.

V ::= variables (including _, the underscore pattern)
A ::= atoms
I ::= integers
K ::= A | I | other constants (e.g. strings, floats)
P ::= K | V | $\{P,\ldots,P\}$ | `[`$P,\ldots,P$`|`P`]`
E ::= K | V | $\{E,\ldots,E\}$ | `[`$E,\ldots,E$`|`E`]` | `[`E`||`P`<-`E`]` | P `=` E |
$E \circ E$ | `(`E`)` | E`(`$E,\ldots,E$`)` |

```
case E of
    P when E -> E,...,E;
    ⋮
    P when E -> E,...,E
end
```

F ::= A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$;
⋮
A`(`$P,\ldots,P$`)` when E `->` $E,\ldots,E$.

Fig. 3. The used Erlang syntax subset

– It does not contain those expression types which can be handled in the same way as one from the presented expressions. For example, the if and try construct can be handled similar as case expressions.
– Those language constructs which are not used to build the data dependency graph also left out from the model. For example, the attributes of an Erlang module do not hold relevant information in the meaning of data dependency.

4 Behaviour Dependency Graph

The most natural way to represent the impact of a change is a graph. To propagate dependency information we build a behaviour dependency graph (BDG).

4.1 The Representation of the Erlang Programs

To build the Erlang dependency graph we use the semantic program graph of RefactorErl. RefactorErl constructs the syntax tree representation of the source code and extends it with static semantic and lexical information. In RefactorErl each expression and pattern node is identified uniquely, we use these nodes as a base of the dependency graph, and the new edges represent the dependency information among them. While constructing the dependency graph we traverse the semantic graph, we take information from the graph, i.e. the structure of the syntax tree (expressions are attached to corresponding code parts), semantic information (the binding structure of the variables, the function calls are linked to the definition of the function, etc). Just those syntactic nodes (mainly the expressions) appear in the dependency graph which are relevant in dependency propagation.

4.2 Dependency Information

All the dependency information is represented in the behaviour dependency graph (BDG). The nodes of the graph are the expressions and patterns from the Erlang source code, the edges of the graph are representing dependency information. There are different kinds of dependency information, that is represented with labeled edges in the graph ($n_1 \xrightarrow{label} n_2$, where n_1 and n_2 are nodes of the graph). The different kinds of dependency edges are the followings:

Definition 1.

- **Data flow edges** – represent data flow between two nodes. There are different kinds of data flow information [7]:
  - Flow edges – $n_1 \xrightarrow{f} n_2$, represents that the result of n_2 can be a copy of the result of n_1. They value exactly the same, and changing the value of n_1 results the same change in the value of n_2.
  - Constructor edges – $n_1 \xrightarrow{c_i} n_2$, represents that the result of n_2 can be a compound value that contains n_1 as the ith element
  - Selector edges – $n_1 \xrightarrow{s_i} n_2$, represents that the result of n_2 can be the ith element of the n_1 compound data
- **Data dependency edges** – $n_1 \xrightarrow{d} n_2$, represents that the result of n_2 can directly depend on the result of n_1. Any change in n_1 may result a data or a behaviour change in n_2.
- **Behaviour dependency edges** – $n_1 \xrightarrow{b} n_2$, represents that the behaviour of n_2 can directly depend on the result of n_1. Any change in n_1 may result a behaviour change in n_2.

Note, that in case of constructing a list e is used as an element label, because we can not usually track their indexes([7]).

The change of a data has an impact on the behaviour of those expressions which depend on that data, thus each data dependency edge also represents behaviour dependency:

$$\frac{n_1 \xrightarrow{d} n_2}{n_1 \xrightarrow{b} n_2} \qquad \text{(d-b-rule)}$$

Similar, most of the flow edges propagate the change of the data, thus propagate dependency. Therefore there are nodes in the graph which are linked with multiply edges.

Examples. The following example demonstrate the differences among the edge types.

```
e:
  case X of
     {ok, Result} -> Result + 2;
     _ when is_list(X) -> X
  end
```

There are different kinds of flow edges in case of this case expression: e. The result of the variable `X` simply flows to the tuple pattern: `X` $\xrightarrow{f}$ `{ok, Result}`, then while this pattern is a selector, it selects the elements from the tuple: `{ok, Result}` $\xrightarrow{s_1}$ `ok`, `{ok, Result}` $\xrightarrow{s_2}$ `Result`. The result if this case expression is the result of the last expression in its branches, so the result of the last expressions flow into the case expression: `Result+2` $\xrightarrow{f} e$ and `X` $\xrightarrow{f} e$.

The result of the infix expression `Result+2` depends on the value of its subexpressions: `Result` $\xrightarrow{d}$ `Result+2` and `2` $\xrightarrow{d}$ `Result+2`.

This simple example also represent behaviour dependency information. The behaviour of the case expression is depend on the behaviour of its subexpressions. If the infix expression can not be evaluated then e also can not be evaluated: `Result+2` $\xrightarrow{b} e$.

4.3 Dependency Rules

As it is mentioned before, the dependency graph can be constructed based on the syntax tree and semantic information. The construction rules are summarized in Figures 4 and 5, and the major rules are described in the followings. The notation on the figures are: e is an expression (E), g is a guard expression (E), p is a pattern (P) and f is a function (F).

Variable. The only dependency among the variable bindings and the variable occurrences is the data flow (Figure 4: Variable). It does not hold data dependency, or behaviour dependency information.

Match expression. Figure 4: Match exp. shows that the match expression contains a various number of dependency. The value of the expression e simply copied to the pattern p and to the expression e_0, that represented by flow edges. Each expression depends on the behaviour of its subexpressions, thus the match expression also represents behaviour dependency. The expression $p = e$ binds the value of e to p. In case if the variable p is already bound, then the match expression fails if the value of the variable p and the value of e do not match, so the result of the match expression e_0 may depend on the value of e, thus the match expression contains data dependency.

Infix expressions. The infix expression does not propagate data flow information, rather propagates data dependency information (Figure 4: Infix exp.). The result of an infix expression depends on the result of its subexpressions. If one of the subexpressions can not be evaluated, the infix expression can not be evaluated, so it also propagate behaviour dependency information.

Compound data structures. Beside data flow (flow, constructor and selector edges) information compound data structures (tuples, lists) also hold behaviour dependency information (Figures 4: Tuple exp., List exp. and List gen.). The behaviour of a compound data structure depends on the behavior of its elements, i.e. the expression depends on the behaviour of its subexpression.

	Expressions	Graph edges
(Variable)	p is a binding n is a usage of the same variable	$p \xrightarrow{f} n$
(Match exp.)	e_0: $p = e$	$e \xrightarrow{f} e_0$, $e \xrightarrow{d} e_0$, $e \xrightarrow{b} e_0$ $e \xrightarrow{f} p$
(Pattern)	p_0: $p_1 = p_2$	$p_0 \xrightarrow{f} p_1$ $p_0 \xrightarrow{f} p_2$
(Infix exp.)	e_0: $e_1 \circ e_2$	$e_1 \xrightarrow{d} e_0$, $e_1 \xrightarrow{b} e_0$ $e_2 \xrightarrow{d} e_0$, $e_2 \xrightarrow{b} e_0$,
(Parenthesis)	e_0: (e)	$e \xrightarrow{f} e_0$, $e \xrightarrow{b} e_0$
(Tuple exp.)	e_0: $\{e_1, \ldots, e_n\}$	$e_1 \xrightarrow{c_1} e_0, \ldots, e_n \xrightarrow{c_n} e_0$ $e_1 \xrightarrow{b} e_0, \ldots, e_n \xrightarrow{b} e_0$
(Tuple pat.)	p_0: $\{p_1, \ldots, p_n\}$	$p_0 \xrightarrow{s_1} p_1, \ldots, p_0 \xrightarrow{s_n} p_n$
(List exp.)	e_0: $[e_1, \ldots, e_n \| e_{n+1}]$	$e_1 \xrightarrow{c_e} e_0, \ldots, e_n \xrightarrow{c_e} e_0$, $e_{n+1} \xrightarrow{f} e_0$ $e_1 \xrightarrow{b} e_0, \ldots, e_n \xrightarrow{b} e_0$, $e_{n+1} \xrightarrow{b} e_0$
(List gen.)	e_0: $[e_1 \| \| p \leftarrow e_2]$	$e_1 \xrightarrow{c_e} e_0$, $e_2 \xrightarrow{s_e} p$ $e_1 \xrightarrow{b} e_0$, $e_2 \xrightarrow{b} e_0$
(List pat.)	p_0: $[p_1, \ldots, p_n \| p_{n+1}]$	$p_0 \xrightarrow{s_e} p_1, \ldots, p_0 \xrightarrow{s_e} p_n$ $p_0 \xrightarrow{f} p_{n+1}$
(BIF 1)	e_0: $\mathsf{hd}(e_1)$	$e_1 \xrightarrow{s_e} e_0$ $e_1 \xrightarrow{b} e_0$
(BIF 2)	e_0: $\mathsf{tl}(e_1)$	$e_1 \xrightarrow{f} e_0$ $e_1 \xrightarrow{b} e_0$
(BIF 3)	I is constant, e_0: $\mathsf{element}(I, e_1)$	$e_1 \xrightarrow{s_I} e_0$ $e_1 \xrightarrow{b} e_0$

Fig. 4. Static behaviour dependency graph generation rules

Conditional expressions. The behaviour of a conditional expression, like each complex expression, depends on its subexpressions. Thus each subexpression is linked to the expression with a behaviour dependency edge (Figure 4: Case exp.). For example, the case-expression depends on the behaviour of the expressions of its clauses, because an exception in these expressions propagate an exception into the case-expression.

Function calls. A function call similar to complex expressions, depends on its arguments, but it also depends on the body of the referred function (Figure 5: Fun. call 1). An exception from the body of the function has an impact on the function call expression, too. The result of an actual parameter flows into the corresponding formal parameter of the function, and the return value of the

	Expressions	Graph edges
(Case exp.)	e_0: case e of p_1 when $g_1 \to e^1_1,\dots,e^1_{l_1}$; $\vdots$ p_n when $g_n \to e^n_1,\dots,e^n_{l_n}$ end	$e \xrightarrow{f} p_1,\dots,e \xrightarrow{f} p_n$ $e^1_{l_1} \xrightarrow{f} e_0,\dots,e^n_{l_n} \xrightarrow{f} e_0$ $e \xrightarrow{d} e_0,\ e \xrightarrow{b} e_0$ $e^1_1 \xrightarrow{b} e_0,\dots,e^1_{l_1} \xrightarrow{b} e_0$ $\vdots$ $e^n_1 \xrightarrow{b} e_0,\dots,e^n_{l_n} \xrightarrow{b} e_0$ $g_1 \xrightarrow{b} e_0,\dots,g_n \xrightarrow{b} e_0$
(Fun. call 1)	e_0: f$(e_1,\dots,e_n)$ f/n: f$(p^1_1,\dots,p^1_n)$ when $g_1 \to$ $e^1_1,\dots,e^1_{l_1}$; $\vdots$ f$(p^m_1,\dots,p^m_n)$ when $g_m \to$ $e^m_1,\dots,e^m_{l_m}$.	$e_1 \xrightarrow{b} e_0,\dots,e_n \xrightarrow{b} e_0$ $e^1_{l_1} \xrightarrow{f} e_0,\dots,e^m_{l_m} \xrightarrow{f} e_0$ $e_1 \xrightarrow{f} p^1_1,\dots,e_1 \xrightarrow{f} p^m_1$ $\vdots$ $e_n \xrightarrow{f} p^1_n,\dots,e_n \xrightarrow{f} p^m_n$ $e^1_1 \xrightarrow{b} e_0,\dots,e^1_{l_1} \xrightarrow{b} e_0$ $\vdots$ $e^m_1 \xrightarrow{b} e_0,\dots,e^m_{l_m} \xrightarrow{b} e_0$ $g_1 \xrightarrow{b} e_0,\dots,g_m \xrightarrow{b} e_0$
(Fun. call 2)	e_0: $e(e_1,\dots,e_n)$ e is not constant, or e/n undefined	$e_1 \xrightarrow{d} e_0,\dots,e_n \xrightarrow{d} e_0,\ e \xrightarrow{d} e_0$ $e_1 \xrightarrow{b} e_0,\dots,e_n \xrightarrow{b} e_0,\ e \xrightarrow{b} e_0$

Fig. 5. Static behaviour dependency graph generation rules (cont.)

function (the result of the last expression of the function clause) flows back into the function call expression.

$e(e_1,\dots,e_n)$ is an Erlang function call. In case when e is not a constant, we can not create dependency edges between the application and the function definition. The same situation occurs when the function is not defined in our graph – it is not added to the database(Figure 5: Fun. call 2). We handle that case as a worst case scenario, and we generate data dependency edges among the function call and its subexpressions, because we do not know anything about the body of the function and the way how it uses and transforms the value of its parameters.

Built in functions (BIF). There are some built in function in Erlang which operate similar to the data selectors (Figure 4: BIF). For example the `hd/1` function selects the first elements of the list, or the `element/2` function selects the `I`-th element of a tuple. In these cases we add selector edges to the graph.

When the first parameter of the `element/2` function (`I`) is not a constant, the Function call 2 rule is applied.

4.4 Deriving Dependency Information

To determine the impact of a modification we need indirect/deeper dependency knowledge, thus we should calculate the transitive closure of the graph and traverse that graph. Each edge in the graph represent a dependency in the program, therefore when we want to determine the impact of a change, we have to traverse the graph using the corresponding defined edges.

A dependency relation between two graph nodes ($n_1 \rightsquigarrow n_2$) means the behaviour of n_2 depends on the result/behaviour of n_1, so the change of the value of n_1 may have an impact on n_2. This relation can be computed using the data flow, data dependency and the behaviour dependency edges.

The informal definition of the dependency relation $n_1 \rightsquigarrow n_2$ is that n_2 is an expression in the graph which could be affected by changing the value of n_1. Those nodes from the graph which could be a copy of n_1 are affected by changing the value of n_1, so modifying n_1 could have an impact on them. Therefore the data flow propagate the changes (data-rule).

Consider the following expression: `1+2`. Changing the expression `1` to `atom` results that the expression `1+2` could not be evaluated and that results a runtime error. Then each expression which behaviour depend on the value of `1+2` also could not be evaluated. Therefore when there is data dependency connection between two nodes ($n_1 \xrightarrow{d} n_2$), changing the data in n_1 could have an impact on the behaviour of n_2, and those node which behaviour may depend from n_2, also could alter behaviour from the same data change (b-dep-rule). Data flow and the behaviour dependency edges ($\xrightarrow{b}$) also propagates behaviour dependency among expressions (d-rule, b-rule).

In the followings we formalize the mentioned behaviour dependency relation.

Definition 2. The data flow relation $\overset{d}{\rightsquigarrow}$ is defined as the minimal that satisfies the following rules [7]:

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

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

$$\frac{n_1 \xrightarrow{c_i} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3,\ n_3 \xrightarrow{s_i} n_4}{n_1 \overset{d}{\rightsquigarrow} n_4} \qquad \text{(c-s-rule)}$$

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

Definition 3. The behaviour dependency relation $\overset{b}{\rightsquigarrow}$ is defined as the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{d}{\rightsquigarrow} n_2}{n_1 \overset{b}{\rightsquigarrow} n_2} \qquad \text{(d-rule)}$$

$$\frac{n_1 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightarrow} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_4}{n_1 \overset{b}{\rightsquigarrow} n_4} \qquad \text{(b-rule)}$$

Definition 4. The dependency relation $\rightsquigarrow$ is defined as the minimal relation that satisfies the following rules:

$$\frac{n_1 \overset{d}{\rightsquigarrow} n_2}{n_1 \rightsquigarrow n_2} \qquad \text{(data-rule)}$$

$$\frac{n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_4}{n_1 \rightsquigarrow n_4} \qquad \text{(b-dep-rule)}$$

4.5 Lemmas

In this section some lemmas about the properties of relations $\overset{b}{\rightsquigarrow}$ and $\rightsquigarrow$ are introduced. The detailed proofs of the lemmas are presented in appendix A.

Lemma 1 ($\overset{b}{\rightsquigarrow}$ reflexive)
$n \overset{b}{\rightsquigarrow} n$

Proof. Applying the rules (reflexive) and (d-rule).

Lemma 2 ($\overset{b}{\rightsquigarrow}$ transitive)
$n_1 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

Proof. Applying structural induction on $n_1 \overset{b}{\rightsquigarrow} n_2$ and then structural induction on $n_2 \overset{b}{\rightsquigarrow} n_3$.

Lemma 3 ($\rightsquigarrow$ reflexive)
$n \rightsquigarrow n$

Proof. Applying the rules (reflexive) and (data-rule).

Lemma 4 ($\rightsquigarrow$ transitive)
$n_1 \rightsquigarrow n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$

Proof. Applying case distinction on $n_1 \rightsquigarrow n_2$ and then case distinction on $n_2 \rightsquigarrow n_3$.

Lemma 5 (generalized b-dep-rule)
$n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \rightsquigarrow n_4 \Rightarrow n_1 \rightsquigarrow n_4$

Proof. Applying case distinction on $n_3 \rightsquigarrow n_4$.

Lemma 6 ($\overset{b}{\rightsquigarrow}$ is not symmetrical and is not anti-symmetrical)

Proof. See the counterexamples in appendix A.

4.6 Example

Figures 6 and 7 shows the relevant part of the dependency graphs for the motivation examples.

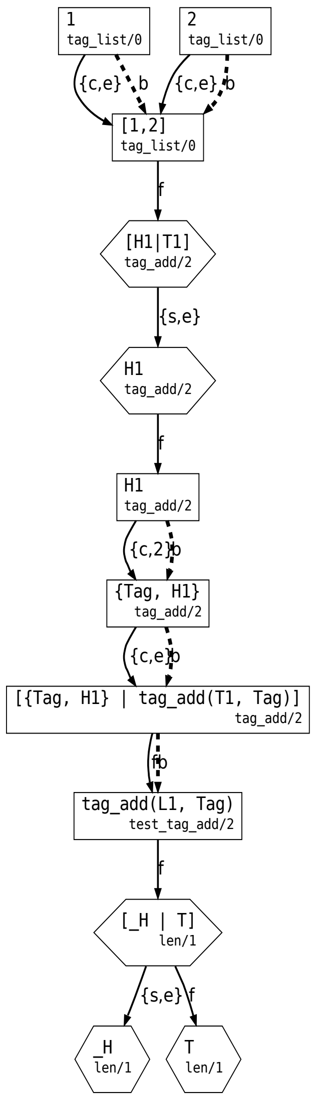

Fig. 6. Dependencies in tag_add

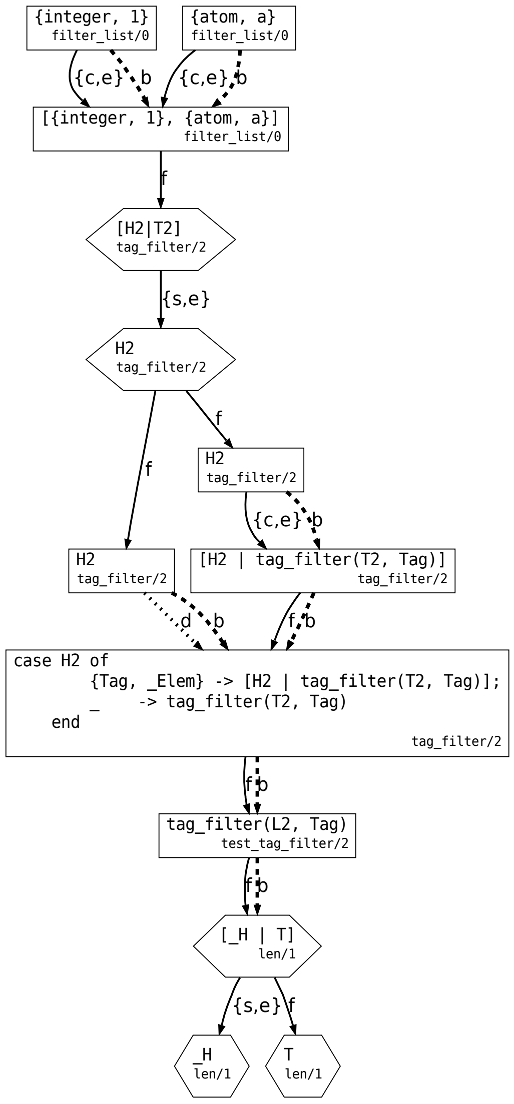
{integer, 1} filter_list/0
{atom, a} filter_list/0
{c,e}
b
[{integer, 1}, {atom, a}] filter_list/0
f
[H2|T2] tag_filter/2
{s,e}
H2 tag_filter/2
H2 tag_filter/2
H2 tag_filter/2
[H2 | tag_filter(T2, Tag)] tag_filter/2
d
case H2 of {Tag, _Elem} -> [H2 | tag_filter(T2, Tag)]; _ -> tag_filter(T2, Tag) end tag_filter/2
tag_filter(L2, Tag) test_tag_filter/2
[_H | T] len/1
_H len/1
T len/1

Fig. 7. Dependencies in tag_filter

The main difference between them is the H2 $\xrightarrow{d}$ `case H2 of ... end` edge, which describes that the result and the behaviour of the case-expression depends on `H2`.

On the left hand side figure, the value `1` simply flows (as a meaning of data flow) into the variable `_H` (in `len/1`). The `len/1` function does not use the value of `_H`, so its return value does not depend on it. On the other figure the value of the tuple `{integer, 1}` flows into the variable `H2` (in `tag_filter/2`). The case-expression depends on the value of `H2`, the value of the case-expression copied to the result of `tag(L1, Tag)` (in `len/1`), so the change of the tuple may have an impact on the `len/1` function.

In summary, if we modify the integer `1` we must not the `test_tag_add/2` test case, but if we modify the tuple `{integer, 1}`, we should run test case `test_tag_filter/2`.

5 Related Work

A methodology for regression test selection in object oriented designs have been already presented in [1]. That methodology represents the designs using the Unified Modeling Language, and gives a formal mapping between design changes and a classification of regression test cases (reusable, retestable, obsolete).

Our model tries to find affected test cases using a graph traversal on the BDG. The BDG adds behaviour edges according to the semantics of the Erlang language to a 0-th order Data Flow Graph (0DFG). Most of the data flow and the behaviour flow edges are specific for Erlang (some of them are discussed in Section 4.3) which do not appears in other languages. Specially, the behaviour edges manly refers to exceptions (that can arise during the evaluation after a data change) in our model. In the analysis of other languages this kind of edges usually do not appear with a data flow graph, rather just the exception handling constructs are handled in a control flow graph [9]. Our model could be applicable to other strict functional languages to detect the spread of a data change, however when applying it to a lazy language further analysis could be useful.

Estimating the impact of a change in functional programming languages is not really widespread yet, however control flow analysis have been already studied by Shivers [8], but this work applied for optimizing compilers. Data flow analysis already defined for Erlang [7] and successfully applied to module interface upgrade. For Erlang the Control flow analysis successfully applied for improving testing [10].

6 Conclusions and Future Work

In this paper we present a dependency graph to calculate the impact of some modification in an Erlang source code. The base idea behind this is to support the programmers to reduce the number of test cases which should be performed after a refactoring transformations. Therefore we have to propagate the change made by the transformation in a behaviour dependency graph. This graph contains the relevant expression nodes from the syntax tree and data flow, data dependency and behaviour dependency edges. The result of the dependency graph has been illustrated in the motivating example.

The size of the presented graph is linear to the size of the syntax tree. If there are n expression nodes in the syntax tree, the size of the graph is $O(n)$, the size of its transitive closure is maximum $O(n^2)$.

This paper shows a structural algorithm to construct the BDG and a relation to calculate the dependency in that graph. The DFG is already implemented in the RefactorErl system, thus we should add the behaviour edges to that graph and implement the dependency relation. Then we could examine the efficiency of our model.

We can improve the presented solution in different ways. The presented model does not make a distinction among the different calls of a function, and the return value of the function is linked to each function call. The problem with that approach is that when we call the function, we reach the result of other function calls, too. To solve this problem we should store context information about the source, i.e. where is the function called. Therefore, more accurate graph could be generated using 1CFA-s [8] or nCFA-s, as the behaviour dependency edges could be generalized according to the order of the analysis, so labeled edges could be used in our BDG to represent the context information. We note, that our model based on a 0DFG and adds behaviour edges to that graph, and it results less test case subset in our example, but using 1DFG-s could also result less test case subset than using 0DFG-s.

When we generate a BDG, it could grow fast, and could be unnecessary huge. Therefore, we should trim the irrelevant parts from the graph. A possible solution should be to combine the control flow with the result of a call graph. First we can create a call graph part from the change, and then we can should create the data flow and the control flow using the affected functions from the call graph. We can build the whole dependency graph and then trim it (for example with slicing), or just build the smaller graph. Thus we can calculate other analysis and iterative algorithms on smaller graphs.

References

1. Briand, L., Labiche, Y., Soccar, G.: Automating impact analysis and regression test selection based on uml designs. In: 18th IEEE International Conference on Software Maintenance, ICSM 2002 (2002)
2. Ericsson, AB, Erlang Reference Manual, `http://www.erlang.org/doc/reference_manual/part_frame.html`
3. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
4. Fredlund, L.-A.: A Framework for Reasoning about ERLANG code. PhD thesis, Royal Institute of Technology, Stockholm, Sweden (2001)
5. Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Tóth, M., Bozó, I., Király, R.: Modeling semantic knowledge in Erlang for refactoring. In: Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, Cluj-Napoca, Romania, Sp. Issue of Studia Universitatis Babe-Bolyai, Series Informatica, vol. 54, pp. 7–16 (July 2009)
6. Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A.N., Nagy, T., Tóth, M., Király, R.: Building a refactoring tool for erlang. In: Workshop on Advanced Software Development Tools and Techniques, WASDETT 2008, Paphos, Cyprus (July 2008)

7. Lövei, L.: Automated module interface upgrade. In: Erlang 2009: Proceedings of the 8th ACM SIGPLAN workshop on Erlang, pp. 11–22. ACM, New York (2009)
8. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University (1991)
9. Sinha, S., Harrold, M.J.: Control-flow analysis of programs with exception-handling constructs. Technical Report (1998)
10. Widera, M.: Flow graphs for testing sequential erlang programs. In: Proceedings of the ACM SIGPLAN 2004 Erlang Workshop, pp. 48–53 (2004)

A The Detailed Proofs of Lemmas.

Lemma 1 ($\overset{b}{\rightsquigarrow}$ reflexive)
$n \overset{b}{\rightsquigarrow} n$

Proof

$$\begin{array}{cl} n \overset{d}{\rightsquigarrow} n \ \text{(reflexive)} & \\ \models============= & \text{(d-rule)} \\ n \overset{b}{\rightsquigarrow} n & \end{array}$$

□

Lemma 2 ($\overset{b}{\rightsquigarrow}$ transitive)
$n_1 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

Proof. structural induction on $n_1 \overset{b}{\rightsquigarrow} n_2$

a) (base step) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$
   structural induction on $n_2 \overset{b}{\rightsquigarrow} n_3$
   i) (base step) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

$$\begin{array}{cl} n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3 & \\ \models============= & \text{(transitive)} \\ n_1 \overset{d}{\rightsquigarrow} n_3 & \\ \models========= & \text{(d-rule)} \\ n_1 \overset{b}{\rightsquigarrow} n_3 & \end{array}$$

   ii) (induction step) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

$$\begin{array}{cl} n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3 & \\ \models========================== & \text{(induction hypothesis)} \\ n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3 & \\ \models===================== & \text{(b-rule)} \\ n_1 \overset{b}{\rightsquigarrow} n_3 & \end{array}$$

b) (induction step) $n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \overset{b}{\rightsquigarrow} n_3$

$$n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_3$$
$$\models\text{========================}\quad\text{(induction hypothesis)}$$
$$n_1 \overset{b}{\rightsquigarrow} n_4,\ n_4 \overset{b}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3$$
$$\models\text{====================}\quad\text{(b-rule)}$$
$$n_1 \overset{b}{\rightsquigarrow} n_3$$

□

Lemma 3 ($\rightsquigarrow$ reflexive)
$n \rightsquigarrow n$

Proof

$$n \overset{d}{\rightsquigarrow} n \text{ (reflexive)}$$
$$\models\text{============}\quad\text{(data-rule)}$$
$$n \rightsquigarrow n$$

□

Lemma 4 ($\rightsquigarrow$ transitive)
$n_1 \rightsquigarrow n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$

Proof
case distinction on $n_1 \rightsquigarrow n_2$

a) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$
case distinction on $n_2 \rightsquigarrow n_3$
i) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3 \Rightarrow n_1 \rightsquigarrow n_3$

$$n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_3$$
$$\models\text{============}\quad\text{(transitive)}$$
$$n_1 \overset{d}{\rightsquigarrow} n_3$$
$$\models\text{=========}\quad\text{(data-rule)}$$
$$n_1 \rightsquigarrow n_3$$

ii) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3 \Rightarrow n_1 \rightsquigarrow n_3$

$$n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3$$
$$\models\text{========================}\quad\text{(transitive)}$$
$$n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_3$$
$$\models\text{==================}\quad\text{(b-dep-rule)}$$
$$n_1 \rightsquigarrow n_3$$

b) $n_1 \overset{d}{\rightsquigarrow} n_4,\ n_4 \overset{d}{\rightarrow} n_5,\ n_5 \overset{b}{\rightsquigarrow} n_2,\ n_2 \rightsquigarrow n_3 \Rightarrow n_1 \rightsquigarrow n_3$
case distinction on $n_2 \rightsquigarrow n_3$

i) $n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_2,\ n_2 \overset{d}{\leadsto} n_3 \Rightarrow n_1 \leadsto n_3$

$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_2,\ n_2 \overset{d}{\leadsto} n_3$$
$$\models\!= \quad \text{(d-rule)}$$
$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_2,\ n_2 \overset{b}{\leadsto} n_3$$
$$\models\!= \quad (\overset{b}{\leadsto} \text{ transitive})$$
$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_3$$
$$\models\!= \quad \text{(b-dep-rule)}$$
$$n_1 \leadsto n_3$$

ii) $n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_2,\ n_2 \overset{d}{\leadsto} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\leadsto} n_3$
$\Rightarrow n_1 \leadsto n_3$

$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_2,$$
$$n_2 \overset{d}{\leadsto} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\leadsto} n_3$$
$$\models\!= \quad \text{(d-rule)}$$
$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_2,$$
$$n_2 \overset{b}{\leadsto} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\leadsto} n_3$$
$$\models\!= \quad (\overset{b}{\leadsto} \text{ transitive})$$
$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,$$
$$n_5 \overset{b}{\leadsto} n_6,\ n_6 \overset{d}{\to} n_7,\ n_7 \overset{b}{\leadsto} n_3$$
$$\models\!= \quad \text{(d-b-rule)}$$
$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,$$
$$n_5 \overset{b}{\leadsto} n_6,\ n_6 \overset{b}{\to} n_7,\ n_7 \overset{b}{\leadsto} n_3$$
$$\models\!= \quad \text{(b-rule)}$$
$$n_1 \overset{d}{\leadsto} n_4,\ n_4 \overset{d}{\to} n_5,\ n_5 \overset{b}{\leadsto} n_3$$
$$\models\!= \quad \text{(b-dep-rule)}$$
$$n_1 \leadsto n_3$$

□

Lemma 5 (generalized b-dep-rule)
$n_1 \overset{d}{\leadsto} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \leadsto n_4 \Rightarrow n_1 \leadsto n_4$

Proof
case distinction on $n_3 \leadsto n_4$

a) $n_1 \overset{d}{\leadsto} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{d}{\leadsto} n_4 \Rightarrow n_1 \leadsto n_4$

$$n_1 \overset{d}{\leadsto} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{d}{\leadsto} n_4$$
$$\models\!= \quad \text{(b-rule)}$$
$$n_1 \overset{d}{\leadsto} n_2,\ n_2 \overset{d}{\to} n_3,\ n_3 \overset{b}{\leadsto} n_4$$
$$\models\!= \quad \text{(b-dep-rule)}$$
$$n_1 \leadsto n_4$$

b) $n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_5,\ n_5 \overset{d}{\rightarrow} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 \Rightarrow n_1 \rightsquigarrow n_4$

$$\begin{array}{cl}
n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_5,\ n_5 \overset{d}{\rightarrow} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 & \\
\models\!= & \text{(d-b-rule)} \\
n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \overset{d}{\rightsquigarrow} n_5,\ n_5 \overset{b}{\rightarrow} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 & \\
\models\!= & \text{(d-rule)} \\
n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_5,\ n_5 \overset{b}{\rightarrow} n_6,\ n_6 \overset{b}{\rightsquigarrow} n_4 & \\
\models\!= & \text{(b-rule)} \\
n_1 \overset{d}{\rightsquigarrow} n_2,\ n_2 \overset{d}{\rightarrow} n_3,\ n_3 \overset{b}{\rightsquigarrow} n_4 & \\
\models\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!= & \text{(b-dep-rule)} \\
n_1 \rightsquigarrow n_4 &
\end{array}$$

□

Lemma 6 ($\overset{b}{\rightsquigarrow}$ is not symmetrical and is not anti-symmetrical)

Proof. Lets consider the following examples:

Example 1

```
ten()-> 10.
add_ten(X) -> X + ten().
```

$\overset{b}{\rightsquigarrow}$ is not symmetrical if exist two expression in the graph n_1 and n_2 where $n_1 \overset{b}{\rightsquigarrow} n_2$ but *not* $n_2 \overset{b}{\rightsquigarrow} n_1$. If n_1 is the integer `10` from the body of `ten/0` and n_2 is the function call `ten()` in the body of `add_ten/1` then:

$$\begin{array}{cl}
n_1 \overset{b}{\rightsquigarrow} n_1,\ n_1 \overset{b}{\rightarrow} n_2,\ n_2 \overset{b}{\rightsquigarrow} n_2 & \\
\models\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!=\!= & \text{(b-rule)} \\
n_1 \overset{b}{\rightsquigarrow} n_2 &
\end{array}$$

Based on the behaviour dependency graph building rules (Figures 4. and 5.), only two edges start from n_2: a $\overset{b}{\rightarrow}$ and a $\overset{d}{\rightarrow}$ edge, both to the direction of the infix expression `X + ten()`. There is now direct edges or graph paths starting from the expression `X + ten()`, thus does not exist any path from n_2 to n_1, so *not* $n_2 \overset{b}{\rightsquigarrow} n_1$.

Example 2

```
f(0) -> 0;
f(A) when A > 0 -> f(A) - 1.
```

$\overset{b}{\rightsquigarrow}$ is not anti-symmetrical if exist two expression in the graph n_1 and n_2 where $n_1 \overset{b}{\rightsquigarrow} n_2$ and $n_2 \overset{b}{\rightsquigarrow} n_1$. If n_1 is the infix expression `f(A)-10` and n_2 is the function call `f(A)` then both $n_1 \overset{b}{\rightsquigarrow} n_2$ and $n_2 \overset{b}{\rightsquigarrow} n_1$ are true:

$$\begin{array}{c}n_1 \xrightarrow{f} n_2\\ \models\!=\!=\!=\!=\!=\!=\!=\!=\!= \quad \text{(f-rule)}\\ n_1 \overset{d}{\rightsquigarrow} n_2\\ \models\!=\!=\!=\!=\!=\!=\!=\!=\!= \quad \text{(d-rule)}\\ n_1 \overset{b}{\rightsquigarrow} n_2\end{array}$$

$$\begin{array}{c}n_2 \overset{b}{\rightsquigarrow} n_2\ (\overset{b}{\rightsquigarrow}\ reflexive),\ n_2 \xrightarrow{b} n_1,\ n_1 \overset{b}{\rightsquigarrow} n_1\ (\overset{b}{\rightsquigarrow}\ reflexive)\\ \models\!= \quad \text{(b-rule)}\\ n_2 \overset{b}{\rightsquigarrow} n_1\end{array}$$

$n_1 \xrightarrow{f} n_2$ is based on the rule *Fun call 1.* and $n_2 \xrightarrow{b} n_1$ is based on the rule *Infix exp.*

□