Final Pattern Discovery

D2.13

Due date of deliverable: 31st December 2014

Start date of project: October 1st, 2011

Type: Deliverable
WP number: WP2
Task number: T2.4

Responsible institution: ELTE
Editor and editor’s address: M. Tóth, I. Bozó (ELTE-Soft) and T. Kozsik (ELTE)

---

<table>
<thead>
<tr>
<th>Dissemination Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
<td>Public</td>
</tr>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including the Commission Services)</td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including the Commission Services)</td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
</tr>
</tbody>
</table>

Project co-funded by the European Commission within the Seventh Framework Programme
Executive Summary

Task 2.4 aims to develop methodology to automatically discover code fragments that match parallel patterns defined in Work Package 2. In D2.11 we have described the initial pattern discovery that targeted the identification of certain library calls and language constructs. In this report we focus on the characterisation of function definitions that represent computational behaviour equivalent (or similar) to that of the ParaPhrase patterns defined in D2.5.

We have studied and formalised the properties of function definitions that match the ParaPhrase patterns. The pattern discovery mechanism searches for source code fragments that meet the described properties.

We have characterised matching in terms of control-flow, data-flow, and data-dependence information. This description can cope with the different possible syntactical forms that can capture the same semantics, the matching behaviour.
Positioning of Deliverable D2.13

The positioning of this deliverable with respect to other deliverables in the work package can be seen in Figure 1.

![Diagram showing the positioning of deliverable D2.13 with respect to other WP2 deliverables.]

Figure 1: Positioning of deliverable D2.13 w.r.t. other WP2 deliverables
The positioning of this deliverable with respect to other deliverables in other packages can be seen in Figure 2.

Figure 2: Positioning of deliverable D2.13 w.r.t. other deliverables
Contents

Executive Summary ................................................. 1

1 Introduction .................................................. 5

2 Identified Pattern Candidates ................................. 7
  2.1 Map/farm/pipe candidates ............................... 7
  2.2 Reduce candidates ....................................... 9
  2.3 Divide&conquer/search/sort candidates ................. 10
    2.3.1 Sort Pattern ..................................... 11
    2.3.2 Search Pattern .................................. 11
  2.4 Pool/orbit candidates ................................... 12
    2.4.1 Orbit Pattern .................................... 12

3 How to identify pattern candidates? .......................... 13
  3.1 Support analyses to enable pattern discovery .......... 13
    3.1.1 Data Flow Graph ................................ 13
    3.1.2 Control Flow Graph ............................... 14
  3.2 Example: identifying map-like recursive functions ...... 15

4 Conclusion .................................................... 18
Chapter 1

Introduction

In this deliverable we provide the final component of the methodology that enables the discovery of ParaPhrase pattern candidates in Erlang programs. Pattern candidates are code fragments that can be replaced by applications of the parallel patterns developed in Work Package 2 (see D2.5). Pattern discovery aims to identify pattern candidates in the source code by static program analysis techniques.

Pattern candidate discovery is made up of three main components:

- **Analysis of list comprehensions** [5] this component finds and categorises list comprehensions as farm, map or pipe candidates.
- **Analysis of library calls** [5] this component searches for the applications of certain library calls, and categorises them as farm, map or pipe candidates.
- **Analysing the behaviour of Erlang functions** [2] this component scans the source code, and checks each function definition whether it matches a given set of properties, which describes the behaviour of some pattern candidate.

In this deliverable, we focus on the third component. We described the general behaviour of patterns with a set of properties in terms of data-flow, control-flow and dependence information [2]. These properties form the basis of the pattern discovery analysis.

The result of the discovery process (extended with benchmarking based ranking) is presented to the user in the Pattern Candidate Browser [5, 6].

In order to perform pattern discovery, an appropriate representation of the source code is required. We have used and extended the Semantic Program Graph, and the analyser framework, of RefactorErl [1].

This document is structured as follows. In Chapter 2, we explain informally the semantic properties that characterize the map/farm/pipe, the reduce, the divide-and-conquer and the pool/orbit patterns. In Chapter 3, we provide an overview
of the data-flow, control-flow and dependence graphs used by the definition of the properties (Section 3.1) and an example for the formal description of candidate properties (Section 3.2). The example is the property set for identifying map-like recursive functions.

The further details of the formalization of candidate properties can be found in a technical report [2].
Chapter 2

Identified Pattern Candidates

Parallel patterns express structured iteration of certain operations over a given data set or stream, thus we have to find iterative constructs in the language. Since Erlang is a functional programming language, the main component of the advanced pattern discovery is the analysis of those recursive functions that express the required iteration. We have categorised relevant recursive function definitions in Erlang based on their behaviour:

- map-like recursive functions,
- map-like tail-recursive functions,
- foreach-like functions,
- map-like stream-based recursive functions,
- foreach-like stream-based recursive functions,
- pipeline-like functions,
- reduce-like functions,
- divide & conquer-like functions and
- pool-like functions.

2.1 Map/farm/pipe candidates

The farm and the map patterns model embarrassingly stream/data-parallel computations, where a worker function is iterated over the elements of the input. Therefore, in the pattern discovery process we have to identify language constructs that express a form of iteration either over the elements of a data structure (i.e. a list) or an input stream (i.e. received messages).
Pipelines consist of multiple computational stages, and perform well when a large amount of independent data is provided as input. Each data is individually passed through the pipeline, and each stage transforms it. Thus, a pipeline is also an iterative model that applies the same computations (the stages of the pipe) to every data item. Thus, in terms of pattern discovery, pipes are similar to farms and maps: only the computation to be applied on a data item has to further analysed. Once we have identified a recursive function as “map-like”, we can investigate whether it is “pipeline-like” as well.

For any function $f$, the common properties of pattern candidates based on list processing are the following:

- $f$ must be recursive, and must have a base case;
- $f$ should not be of divide-and-conquer style, i.e. it should not recurse multiple times;
- $f$ may have multiple parameters (like in $f(V_1, L, V_2)$), among which there is one, a list $L$, which will be processed element-by-element;
- the expression returned by $f$ must satisfy further requirements:
  - the non-recursive branch(es) (i.e. base case) has to return an empty list;
  - the recursive branch(es) has to return a list, whose head does not depend on the tail of the processed list, and whose tail is the result of the recursive call (where the additional parameters of the function $(V_1, V_2)$ remained the same, and the list parameter is the tail of the original list $L$), so it does not depend on the head of the list $L$.

A special case is when a function over a list is not called for its return value, but rather to its side-effects to take place. When such a foreach-like function is analysed, we can slightly weaken the above condition set; we do not need any restrictions on the return value of the base case. The requirement is that no expressions occurring in the base case may depend on the tail of $L$.

In order to improve performance, programmers may decide to implement iteration with tail-recursion, introducing an additional parameter, the accumulator. When such a tail-recursive function is analysed, we need to set up further requirements on this accumulator $Acc$:

- the accumulator is a list, and it is extended during the recursive call chain – but the added element may depend only on the head of the input list, and may not depend on its tail;
- the base case returns the value of the accumulator;
- in the recursive branch, the last expression to evaluate is a recursive call on the tail of the input list and the new accumulator.
It is also possible that the additional parameters \((V_1, V_2)\) of the recursive function are changed in the recursive call. However, if we want to automatically introduce a skeleton to replace the pattern candidate in such a case, we have to divide the computation into two phases: at first we have to pre-generate the changes of the parameters, zip them together with the input list, and then perform the operation. In some cases this transformation can be done automatically, but in more involved cases manual refactoring might be necessary. Anyway, pattern discovery can identify this pattern candidate.

For any function \(f\), the common properties of pattern candidates based on stream-processing are the following:

- \(f\) must be tail recursive, and must have a base case;
- \(f\) should not recurse multiple times;
- \(f\) may have multiple parameters (like in \(f(V_1, L, V_2)\)), among which there is one, a list \(L\), which will be extended during the recursive calls; this list will serve as an accumulator;
- the expression returned by \(f\) must satisfy further requirements:
  - it has to be the last evaluated expression of a receive-expression branch;
  - the base case has to return \(L\);
  - the last evaluated expression on the recursive branches is the tail recursive call, where the additional parameters of the function \((V_1, V_2)\) remain the same, and the list parameter is extended by one element. This element may depend on the received data (the element of the stream), but must not depend on \(L\).

Again, when a foreach-like stream-based function is analysed, we can weaken slightly this condition set. We do not make any restriction on the return value of the base case, and we do not need an accumulator variable \(L\).

### 2.2 Reduce candidates

The reduce pattern models a data parallel computation where the input items are processed by a commutative and associative binary operation.

The properties of a reduce-like function \(f\) are the following:

- \(f\) must be recursive, and must have a base case;
- \(f\) should not recurse multiple times;
• $f$ may have multiple parameters ($f(V_1, L, V_2)$), among which there is one, a list $L$, which will be processed element-by-element;

• $f$ consumes the list with a binary operator that is commutative and associative;

• the expression returned by $f$ must satisfy further requirements:
  – the base case(s) has to return a value, whose type is the same as the return type of the binary operator;
  – the recursive branch(es) has to return a value that is the result of the application of the binary operator;
  – the first argument of the binary operator may depend on the head of the list $L$, but may not depend on the tail.
  – the second argument of the operator is the result of the recursive function call that depends on the tail of the list, but may not depend on the head of the list $L$;
  – the additional parameters of the function (i.e. $V_1$, $V_2$) remain the same in the recursive call.

If the operation applied on the list elements is not a built-in operator, but a function $g$, further analyses might be necessary, and the above condition set is to be adjusted accordingly. For instance, if $g$ is already used as a reducer operation in another occurrence of the reduce pattern, there is a great chance that it is associative and commutative, and hence $f$ is also an appropriate pattern candidate (a confirmation from the programmer might need to be requested in this case, though).

### 2.3 Divide&conquer/search/sort candidates

The divide and conquer (D&C) pattern describes a computation where a problem is recursively divided into sub-problems (until a given condition), and after solving the sub-problems the sub-solutions are combined to produce the final solution.

A D&C-like function $f$ must satisfy the following conditions:

• $f$ must be recursive, and must have a base case;

• $f$ may have multiple parameters (line in $f(V_1, L, V_2)$), among which there is one, a list, $L$;

• $f$ has multiple recursive calls in its body:
  – it may contain an execution path that contains multiple recursive calls,
- it may contain an expression that expresses the iterative evaluation of a recursive function call; this expression can be
  * a list comprehension which calls \( f \) in the head;
  * a \texttt{lists:map/2} call, whose first parameter is a reference to \( f \);
  * a call to a map-like recursive function \( g \), where every recursive execution path of \( g \) contains a function call to \( f \);
- these multiple recursive calls are data independent;
- the parameters of the recursive call are originating from the list parameter \( L \) of the function, so they must be a sub-list or a transformed sub-list of \( L \) produced by the divide function \( \beta \);
- the return value of the function \( f \) is the result of a combine function/expression \( \alpha \);
- \( \alpha \) may depend on the \( V_1 \) and \( V_2 \) parameters of \( f \), and may depend on the result of the recursive function applications.

There are two special cases for divide-and-conquer.

### 2.3.1 Sort Pattern

The sort pattern takes an input list as an argument, and produces the sorted result. It may be implemented in a D&C style, therefore we can discover it based on the condition set explained above.

### 2.3.2 Search Pattern

The search pattern looks for those elements of a list that satisfy a given condition. This can also be implemented in a D&C style, so the above condition set is able to detect it.

### 2.4 Pool/orbit candidates

The pool pattern models a behaviour when a set of values from a pool is processed by an “evolution” function, and the created values are inserted back to the pool until a given condition (termination condition) holds [4].

A pool-like function \( f \) must satisfy the following conditions:
- \( f \) must be tail recursive, and must have a base case;
• $f$ may have multiple parameters (like in $f(V_1, \text{Pool}, V_2)$), among which there is one, a list $\text{Pool}$, representing the pool;

• $f$ has an expression in its body that expresses the iterative evaluation of the “evolution function”:
  
  – a list comprehension, or
  
  – a lists:map/2 call, or
  
  – a call to a map-like recursive function;

• the parameter of the iteration originates from the list parameter $\text{Pool}$ of the function, so it must be a sub-list or a transformed sub-list of $\text{Pool}$ produced by a selection function $\alpha$;

• the result of the iteration is possibly filtered by a function $\beta$;

• the return point of the function $f$ is:
  
  – the $\text{Pool}$ on the base branch;
  
  – a recursive call to $f$ on the recursive branches (e.g. $f(V_1, \text{NewPool}, V_2)$) where the parameters $V_1, V_2$ are unchanged, and a $\text{NewPool}$ parameter represents the new pool; the new pool may depend on the original parameters of the function, on the selected elements produced by $\alpha$, and on the filtered elements produced by $\beta$.

2.4.1 Orbit Pattern

The orbit pattern models an iterative construction of a set by a few generator functions, until the generation reaches the transitive closure of the initial set. The orbit pattern can be implemented in terms of the pool pattern, therefore the general pool identification algorithm can recognise it as a pool pattern.
Chapter 3

How to identify pattern candidates?

We have characterised the behaviour of patterns in terms of syntactic and static semantic information (such as control-flow, data-flow and dependence information). Pattern discovery takes these properties of the patterns, and scans the source to find functions that correspond to the given properties of patterns. We have categorised the function to be map-like, pipeline-like, divide&conquer-like etc. The properties are formally defined in [2]. As an example, in Section 3.2 we present the characterisation of map-like recursive functions that can be identified as farm/map pattern candidates.

3.1 Support analyses to enable pattern discovery

To successfully implement pattern discovery, an appropriate source code representation and thorough static semantic analyses are required. We use the existing framework of the static source code analysis and transformation tool for Erlang, RefactorErl. The Semantic Program Graph model (SPG) of the tool enables us to search for certain entities in the source code (functions and function calls, list comprehensions etc), although further analyses had to be developed to check the semantic conditions related to the discovery. Besides the SPG we use a Data Flow Graph and a Control Flow Graph as well, and calculate dependencies based on the information provided by those graphs.

3.1.1 Data Flow Graph

The inter-procedural Data Flow Graph of an Erlang program is a labelled directed graph represented by the $DFG = (N, L)$ pair, where $N$ is the set of nodes con-
taining the subexpressions of the analysed program, and \( L \) is the set of graph edges representing the relation that the value of an expression can flow to an other expression, or can be a part of it.

For each \((u, l, v) \in L\) edge, exactly one of the following statements holds:

- \( l = \text{flow} \), and the value of the expression \( u \) flows to the expression \( v \);
- \( l = c_i \), \( u \) is a tuple, and \( u \) is the \( i \)th element of \( v \);
- \( l = s_i \), \( u \) is a tuple pattern, and \( v \) is the \( i \)th element of the pattern;
- \( l = c_h \), \( v \) is a list, and \( u \) is an element of the list;
- \( l = s_h \), \( u \) is a list pattern, and \( v \) is element of the pattern;
- \( l = c_t \), \( v \) is a list, and \( u \) is the tail of the list;
- \( l = s_t \), \( u \) is a list pattern, and \( v \) is the tail of the pattern;
- \( l = d \), i.e. the value of \( v \) depends on the value of \( u \).

To calculate whether the value of a certain expression can reach another expression, we have defined the data flow reaching relation \( \Rightarrow \) and the so-called compact data flow reaching relation \( \Rightarrow_c \).

We also define the dependence relation \( \Rightarrow_d \) over the Dependence Graphs (DG) of Erlang programs [3]. We calculate the DG based on the control and data flow graphs by eliminating the unnecessary sequencing from the CDG, and adding the data relation based on the DFG.

### 3.1.2 Control Flow Graph

The inter-procedural Control Flow Graph of an Erlang program is a labelled directed graph represented by the \( \text{CFG} = (V, E) \) pair, where \( V \) is the set of nodes containing the subexpressions of the analysed program and some special nodes; and \( E \) is the set of graph edges representing the relation “an expression evaluated after another expression”.

For each \( v \in V \), exactly one of the following statements holds:

- \( v \) is a subexpression of the program;
- \( v \) is a special node representing the starting point of a function \((\text{start}_f)\);
- \( v \) represents the end of a function evaluation \((\text{end}_f)\);
- \( v \) represents the call of a function \((\text{call}_f)\);
- \( v \) represents the return of a function call \((\text{ret}_f)\);
For each \((u, l, v) \in E\) edge, exactly one of the following statements holds:

- \(u\) and \(v\) represent subexpressions, and \(v\) is evaluated directly after \(u\) when
  the condition \(l\) holds;
- \(u = \text{call}_f\) represents the call of a function, and \(v = \text{start}_f\) is the starting
  point of the same function;
- \(u = \text{end}_f\) represents the end of a function evaluation, and \(v = \text{ret}_f\) is the
  return point of the call of the same function;

Here we use the notation \(f\) to identify a function. In Erlang it is represented by a
module name, function name and arity triple. If there is no condition between the
evaluation of the two expressions, then the label is empty.

\(EP(v)\) denotes the set of execution paths in the \(CFG\) starting from the node \(v \in V\).

## 3.2 Example: identifying map-like recursive functions

A map-like function \(f\) should have a list parameter, and return a list. The head of
the returned list may depend on the head of the input list, but may not depend on
its tail. Similarly, the tail of the returned list may not depend on the head of the
input list: it should simply be the result of a recursive call to \(f\) on the tail of the
input list. The function may have additional parameters, but all these parameters
must be passed to the recursive call unchanged.

\[
f(\text{Parameter}, \text{List}) \to \begin{array}{l}
\text{case List of} \\
[] \to []; \\
[\text{Head}|\text{Tail}] \to \begin{array}{l}
\text{X} = \ldots \text{Head} \ldots, \\
[\text{X} | f(\text{Parameter}, \text{Tail})]
\end{array}
\end{array}
\]

There are many syntactic forms that satisfy these requirements, one is shown above.
Note that \(f\) need not call itself directly, we allow mutually recursive functions as
well.

Now let us investigate these requirements in a more formal way. We propose con-
servative rules to use for identifying map-like functions. Consider a function \(f\).
We will rely on the control flow and data flow graphs presented in section 3.1
built using \(f\) as the entry point. Thanks to the inter-procedural analyses, mutually
recursive functions are also handled properly.

1. \(f\) must be recursive: the inter-procedural CFG must contain a directed path
   from the starting node of \(f\) to a \text{call}-node of \(f\).

\[\exists p \in EP(\text{start}_f): \text{call}_f \in p\]
2. The definition of \( f \) must have a base case: there is a directed path from the starting node of \( f \) to the ending node of \( f \) which does not include a call-node of \( f \).

\[
\exists p \in EP(\text{start}_f) : (\text{call}_f \notin p) \land (\text{end}_f \in p)
\]

3. \( f \) should not be of divide-and-conquer style, in the sense that it should not recurse multiple times: on each execution path, after a recursive call to \( f \) has returned, no new call to \( f \) may occur.

\[
\forall p \in EP(\text{ret}_f) : \text{call}_f \notin p
\]

4. \( f \) may have multiple parameters, among which there is one, a list, which will be processed element-by-element. Therefore, we assume that the definition of \( f \) is provided with a single clause, the head of which is \( f(V_1, L, V_2) \), where parameter \( L \) corresponds to the list consumed by \( f \). (If \( f \) is defined with multiple clauses, we can meld those into a single clause by introducing an artificial case-expression that contains the pattern matching and the guards of the original function clauses.) We can define the semantic functions \( \text{Head}(L) \) and \( \text{Tail}(L) \) based on the data-flow graph as follows.

\[
\text{Head}(L) = \{ n \in N \mid \exists n' \in N : L \rightarrow^0 n', n' \rightarrow^s n \}
\]

\[
\text{Tail}(L) = \{ n \in N \mid \exists n' \in N : L \rightarrow^0 n', n' \rightarrow^s n \}
\]

5. The expression returned by \( f \) must satisfy further properties. Regarding the different execution paths of \( f \), we can define the set \( R \) of return expressions by collecting the last evaluated expression on all paths before reaching the ending node of \( f \). The last evaluated expression is connected to the ending node by an edge in the CFG.

\[
R = \{ \varrho \mid (\varrho, l, \text{end}_f) \in E \text{ (for some label } l) \}
\]

For all return expression \( \varrho \in R \), the following requirements must hold.

(a) If \( \varrho \) is on a non-recursive execution path, then \( \varrho \) must evaluate to an empty list, i.e. only an empty list constructor is in the compact data-flow reaching relation with \( \varrho \).

\[
\forall n \in N : n \rightarrow^0 \varrho \implies n = []
\]

(b) If \( \varrho \) is on a recursive path, then \( \varrho \) must evaluate to a non-empty list, i.e. in the compact data-flow reaching only expressions in the syntactic form \([\chi \mid \tau] \) or \([\chi] + \tau \) (where \( \chi \) and \( \tau \) are arbitrary expressions).

\[
\forall n \in N : n \rightarrow^0 \varrho \implies \left( \exists \chi, \tau : n = [\chi \mid \tau] \right) \lor
\left( \exists \sigma, \tau : (n = \sigma + \tau) \land (\forall n' \in N : n' \rightarrow^0 \sigma \implies \exists \chi : n' = [\chi]) \right)
\]
(c) With respect to the above condition, we set out further requirements on any possible \( \chi \) and \( \tau \).

i. \( \chi \) depends on \( L \) only through \( \text{Head}(L) \), namely: for each \( L \xrightarrow{d} \chi \) path in the dependence graph, there exists \( n \in \text{Head}(L) \) such that the path is a concatenation of paths \( L \xrightarrow{d} n \) and \( n \xrightarrow{d} \chi \).

ii. \( \tau \) is the result of the recursive call of \( f \), where the parameters of the call are \( V_1 \), the tail of \( L \), and \( V_2 \), namely \( \exists n \in N, \forall n' \in N: \)

\[
\begin{align*}
n &= f(\bar{\nu}_1, \alpha, \bar{\nu}_2) \land V_i \xrightarrow{\text{of}} \bar{\nu}_i \ (i = 1, 2) \land \\
&(n' \xrightarrow{\text{of}} \alpha \Rightarrow n' \in \text{Tail}(L)) \land n \xrightarrow{\text{of}} \tau.
\end{align*}
\]
Chapter 4

Conclusion

In this delivery, we have shown a categorisation and characterisation of Erlang recursive function definitions that represent behaviour that is similar (or equivalent) to that of the ParaPhrase patterns defined in [4].

In Chapter 2 we gave an overview of the properties of the identified functions.

In Chapter 3 we presented the used source code representations and semantic analyses that are applied to precisely define the conditions given in Chapter 2. As an example, we have detailed the definition of map-like recursive functions.

The result of this delivery is exploited by WP4, where the ParaPhrase Refactoring Tool for Erlang (PaRTE) uses the result of the pattern discovery analysis as an input for the semi-automatic parallelization.
Bibliography


