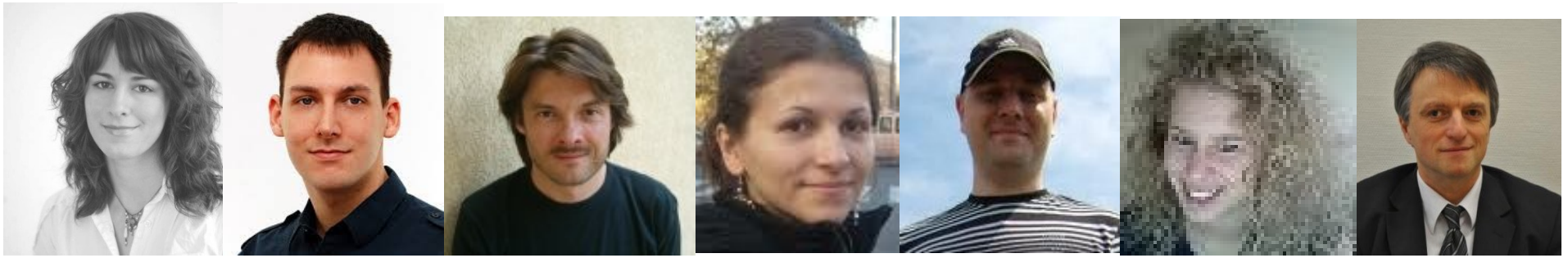
Where shall I parallelize?



Judit Kőszegi, Dániel Horpácsi, Tamás Kozsik, Melinda Tóth, István Bozó, Viktória Fördős, Zoltán Horváth

Eötvös Loránd University
and ELTE-Soft Kft.
Budapest, Hungary

Erlang User Conference
Stockholm, 9-10 June, 2014

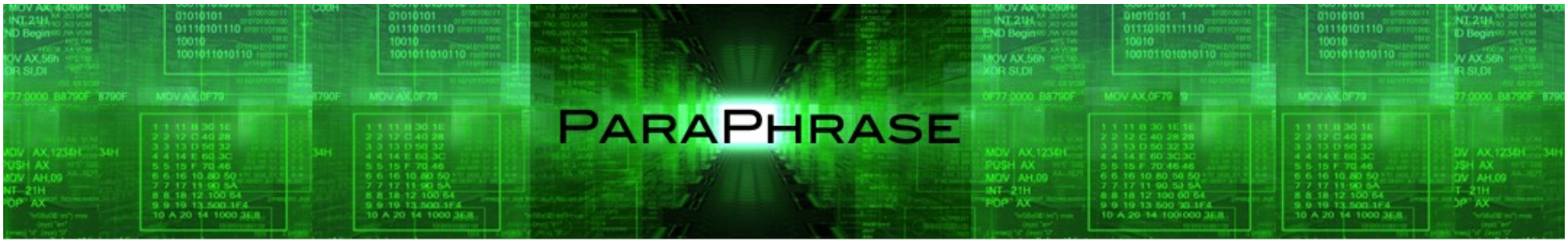
Motivation

- Highly heterogeneous mega-core computers
- Performance and energy
- Think parallel
  - High-level programming constructs
  - Deadlocks etc. eliminated by design
  - Communication packaged/abstracted
  - Performance information is part of design
- Restructure legacy code

Where shall I parallelize?

Tool to...

- find parallelizable code
- help making decisions
- reshape the code
- introduce parallelism



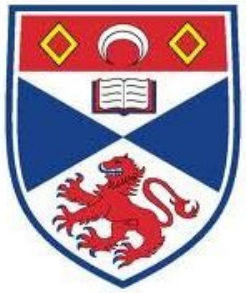
Parallel Patterns for Adaptive Heterogeneous Multicore Systems

ICT-288570 2011-2014 €4.2M budget

13 Partners, 8 European countries

http://www.paraphrase-ict.eu/

@paraphrase_fp7



University of St Andrews

Cloud Competency Center

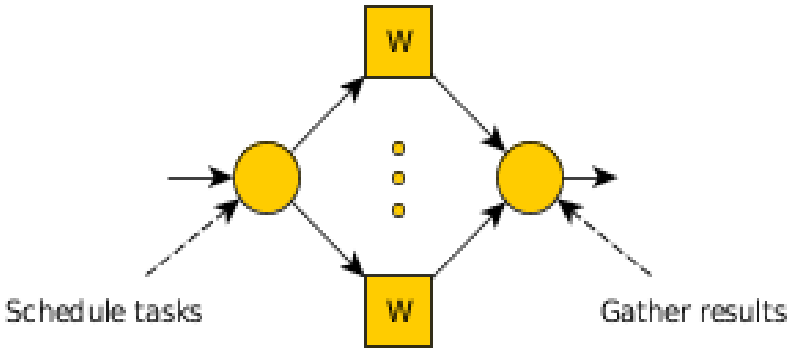
AGH University PL

Universität Stuttgart

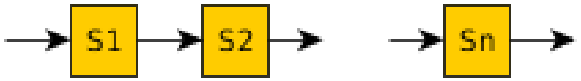
- Programmability of heterogeneous parallel architectures
- Structured design and implementation of parallelism
- High-level parallel patterns

  map(reduce) task farm pipeline
  divide&conquer orbit stencil
- Dynamic (re)mapping on heterogeneous hardware
- C++/FastFlow and Erlang

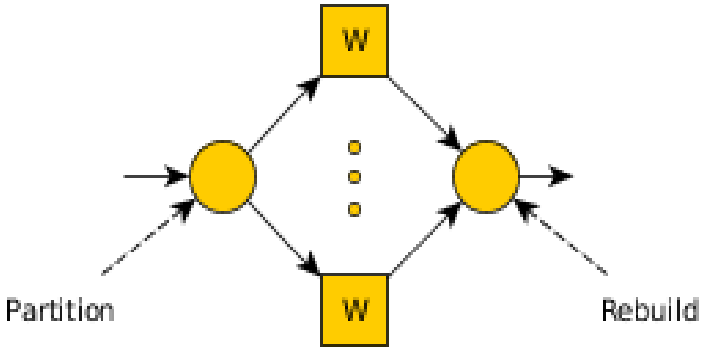
Farm



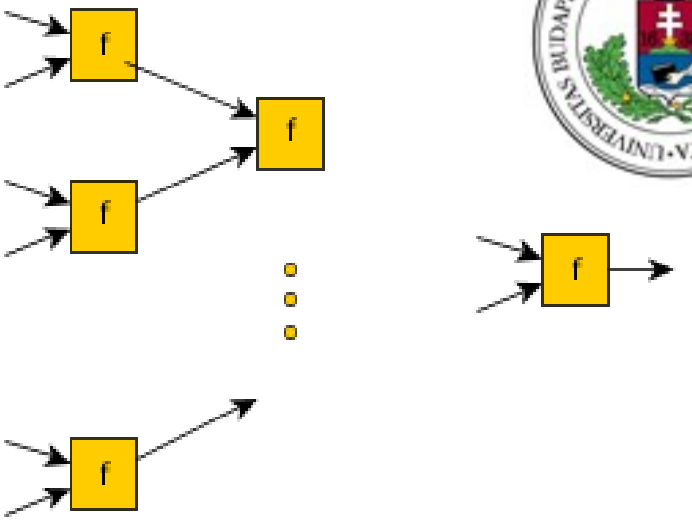
Pipeline



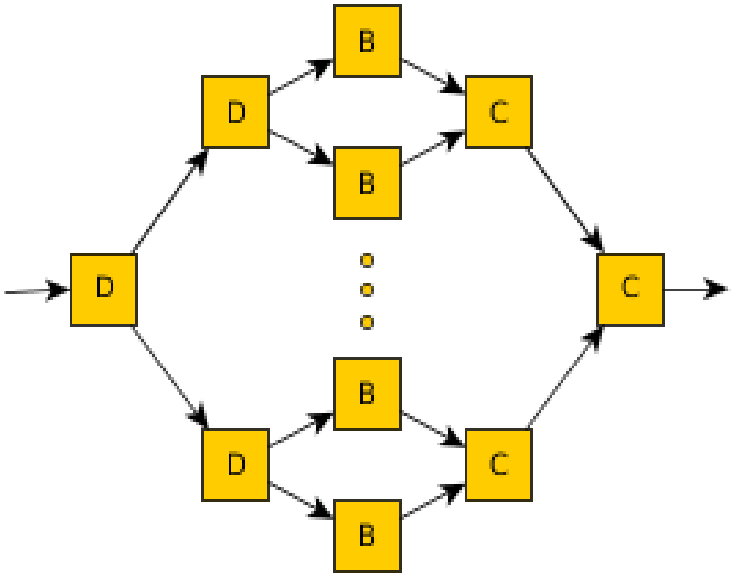
Map



Reduce



Divide&Conquer



- Identify (strongly hygienic) Components
- Patterns of parallelism
- Structure the components into a parallel program
  - Turn the patterns into concrete (skeleton) code
  - Take performance, energy etc. into account
- Restructure if necessary
- Use a refactoring tool!

Christopher Brown's talk

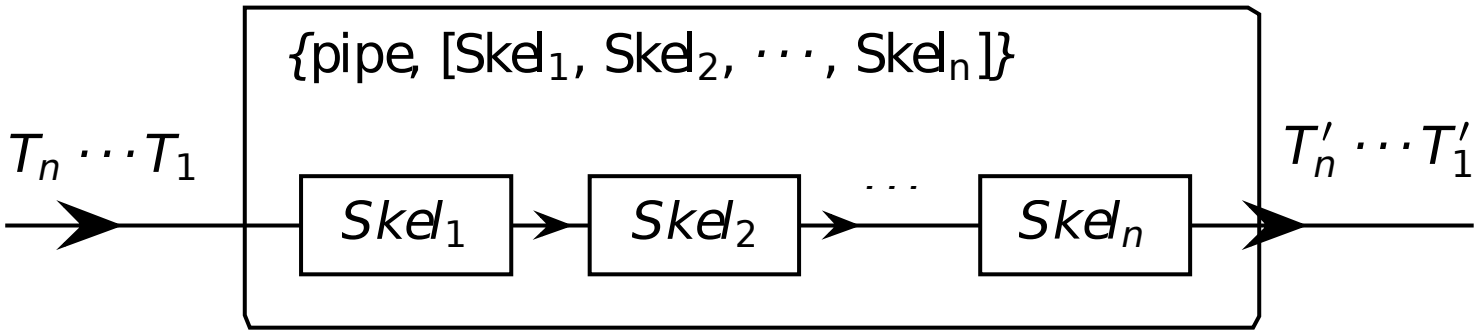
Bridging the Divide:
A New Tool-Supported Methodology for Programming Heterogeneous Multicore Machines

- *Skel* library for Erlang (and C++)
- CPU/GPU systems
- Refactoring tool support (Wrangler)

```
OutputItems =
    skel:do(Skeleton, InputItems)
```

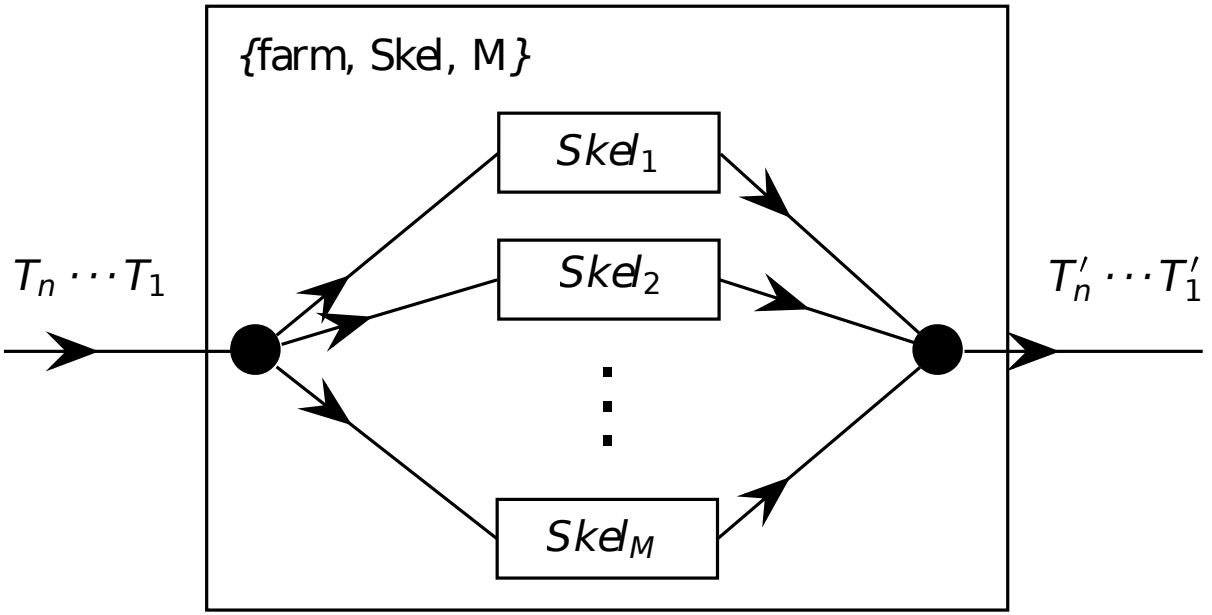
Pipeline


```
skel:do(
   [{pipe,[Skel1, Skel2,..,SkelN]}],
   Inputs
   )
```



Task farm

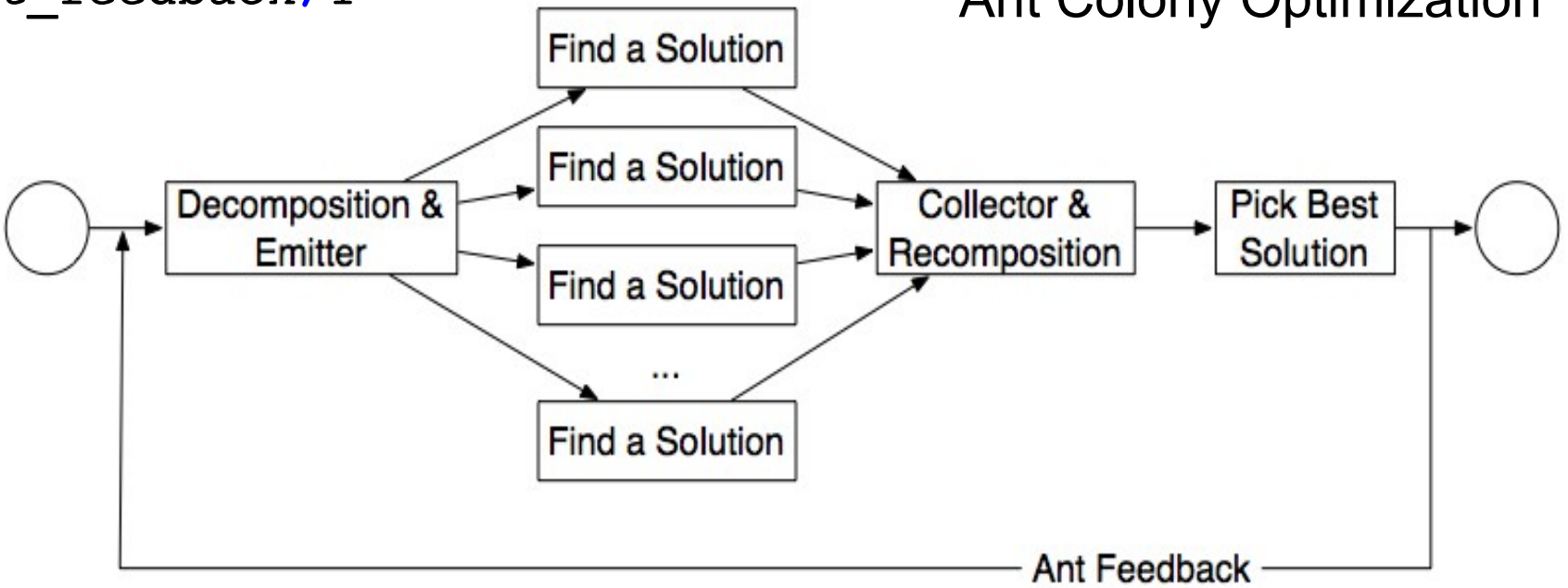
```
skel:do([{farm, Skel, M}], Inputs)
```



Nesting skeletons

```
{ feedback,
  [{pipe, [ {farm, [{seq, fun find_solution/1}], NrW},
            {seq, fun pick_best/1}
          ]
  }],
  fun ant_feedback/1
}
```

Ant Colony Optimization



Example


```
mul(Rows,Cols) ->
    [  [ dotp(Row,Col) || Col <- Cols ]
      || Row <- Rows
    ].

dotp(Row,Col) ->
    lists:sum(
       lists:zipwith(
          fun erlang:'*'/2, Row, Col)).
```

Our goal

Develop tool to...

- Find pattern candidates
- Rank and suggest
- Shape

=> introduce skeletons

Farm


```
mul(Rows,Cols) ->
    [  [ dotp(Row,Col) || Col <- Cols ]
      || Row <- Rows
    ].
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
mul(Rows,Cols) ->
    skel:do( [{ farm, [{seq,
       fun(Row) ->
         [ dotp(Row,Col) || Col <- Cols ] end
       }], 16}], Rows ).
```

PaRTE

ParaPhrase Refactoring Tool for Erlang

Demo!

Expectations

PaRTE...

- changes the way you think about parallelism;
- completely redesigns your code;
- shows all places to introduce parallelism;
- predicts speedup dead exactly;
- is safe and automatic.

Expectations

PaRTE...

- changes the way you think about parallelism;
- completely redesigns your code;
- shows all places to introduce parallelism;
- predicts speedup dead exactly;
- is safe and automatic.

FALSE!

Realistic expectations

PaRTE...

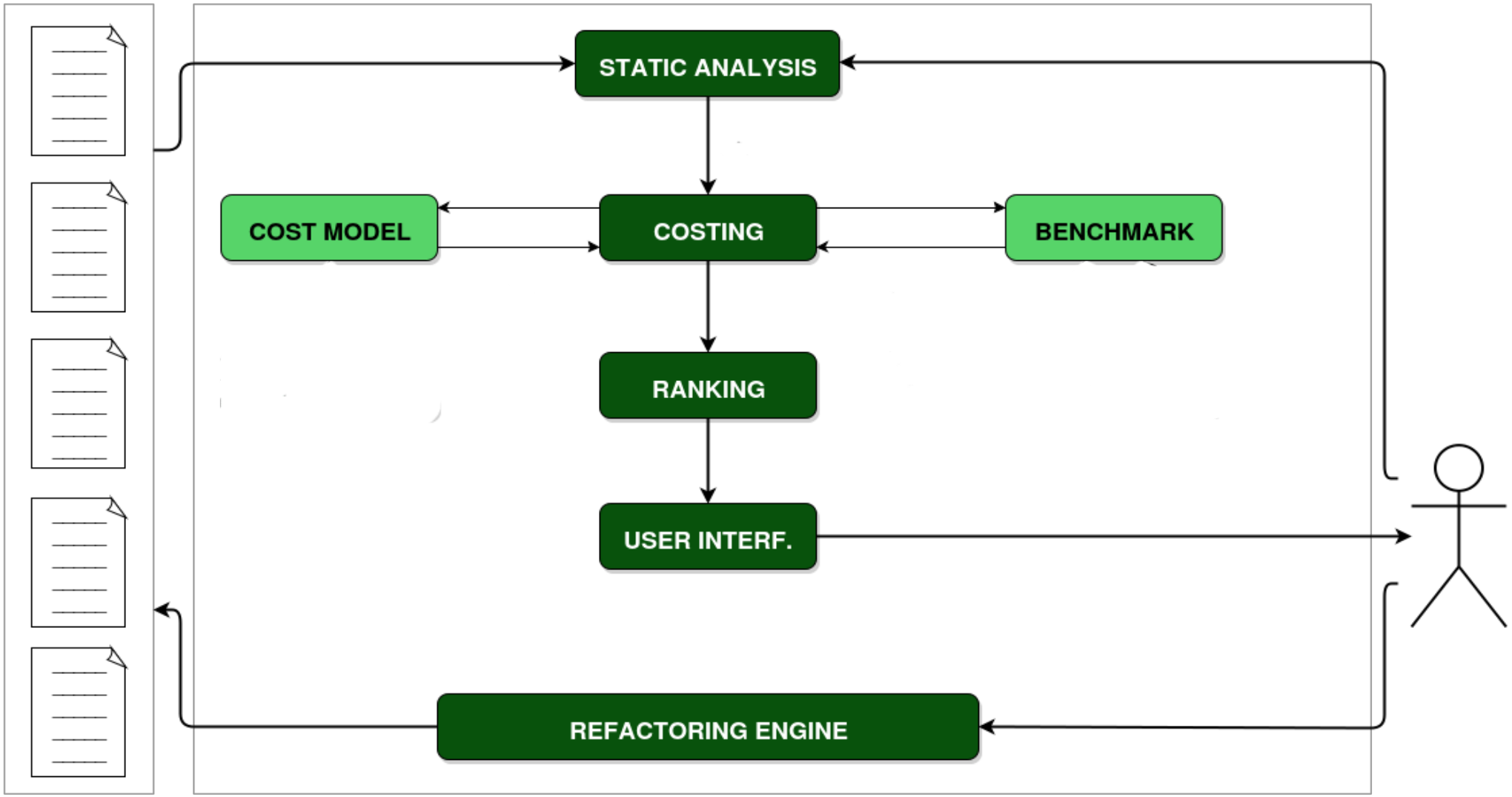
- can find many places to introduce parallelism;
- gives fair speedup predictions;
- works effectively with a smart programmer;
- offers performance gains with small effort.

Key idea

PaRTE can predict speedup by

- measuring sequential execution time on random input, and
- estimate parallel execution time.

Big picture

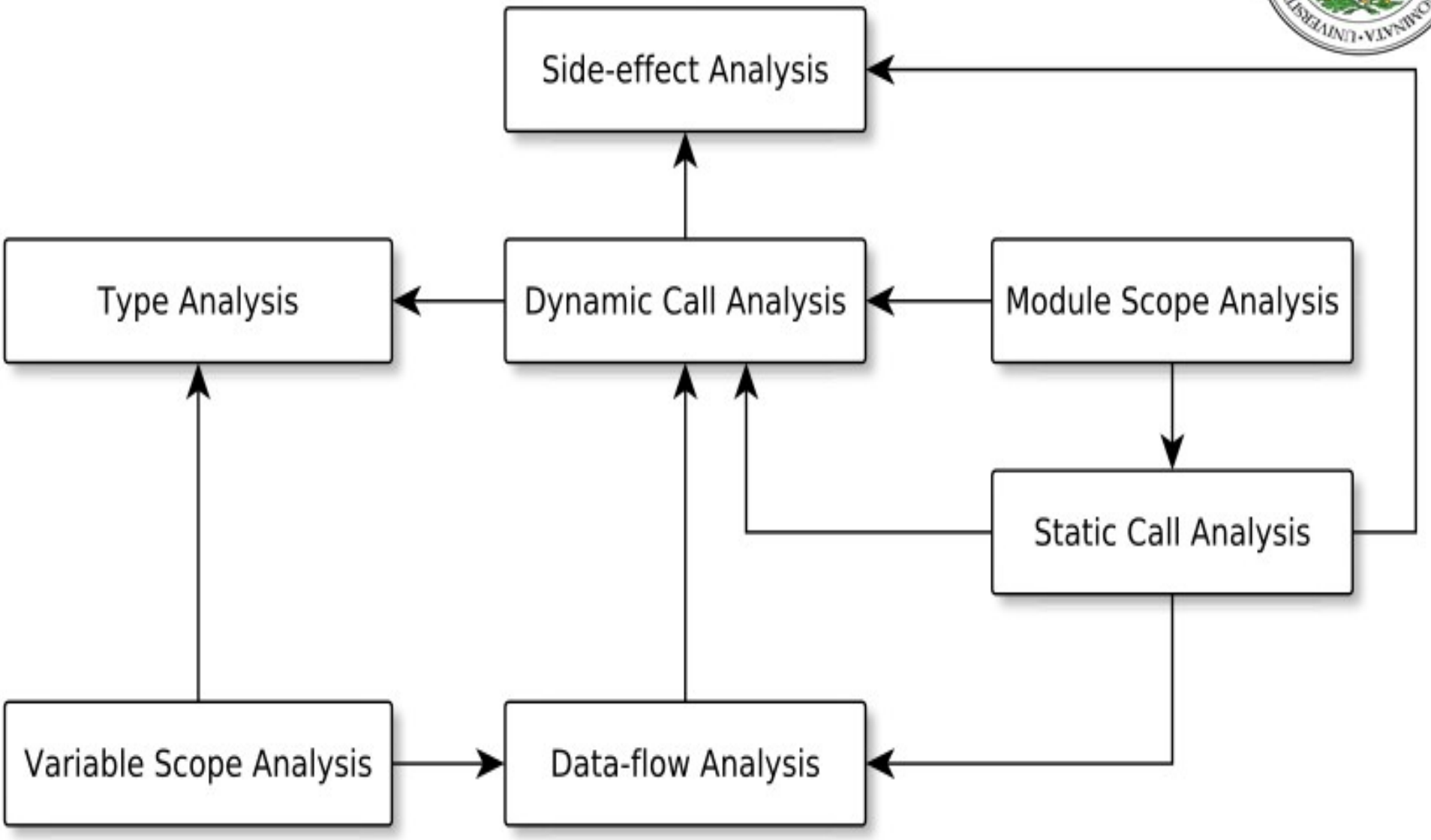

STATIC ANALYSIS
COST MODEL
COSTING
BENCHMARK
RANKING
USER INTERF.
REFACTORING ENGINE

Pattern candidate discovery

- Syntactic (& semantic) information
  - List comprehensions
  - Recursive function definitions
- Side conditions
- Heuristics

```
mul(Rows,Cols) ->
    [  [ dotp(Row,Col) || Col <- Cols ]
     || Row <- Rows
    ].
```

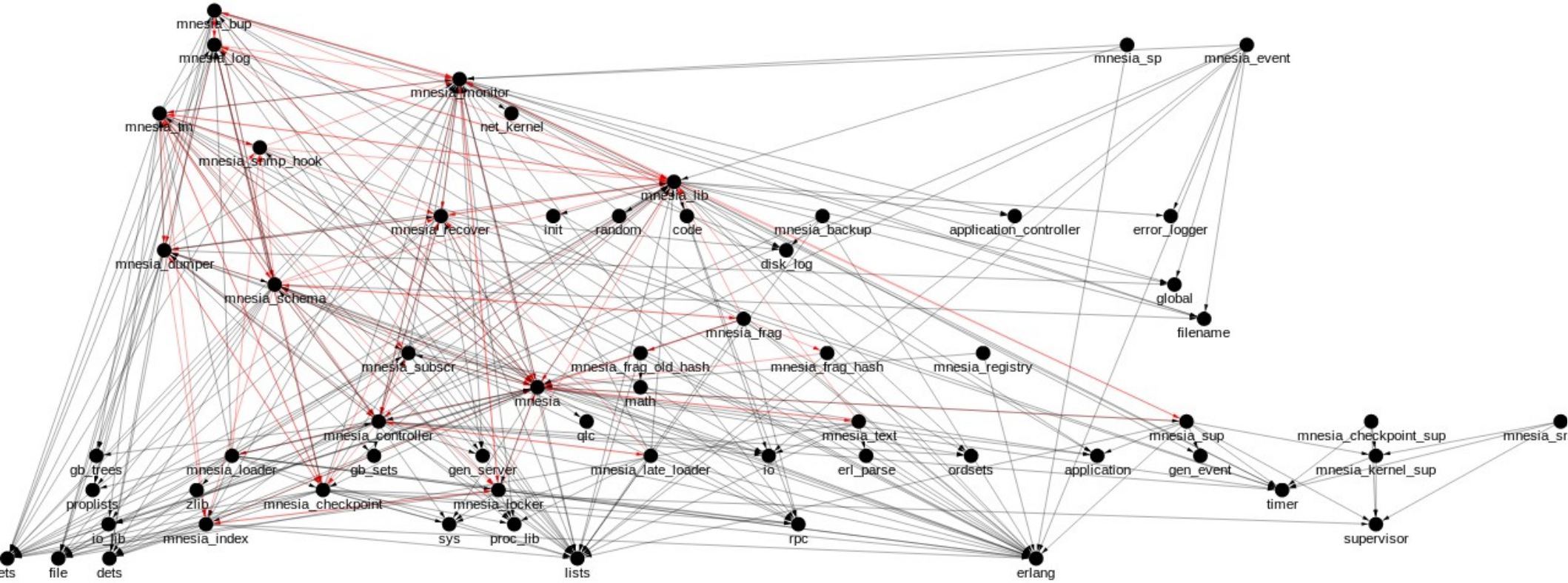
Analyses


Side-effect Analysis
Type Analysis
Dynamic Call Analysis
Module Scope Analysis
Static Call Analysis
Variable Scope Analysis
Data-flow Analysis

RefactorErl

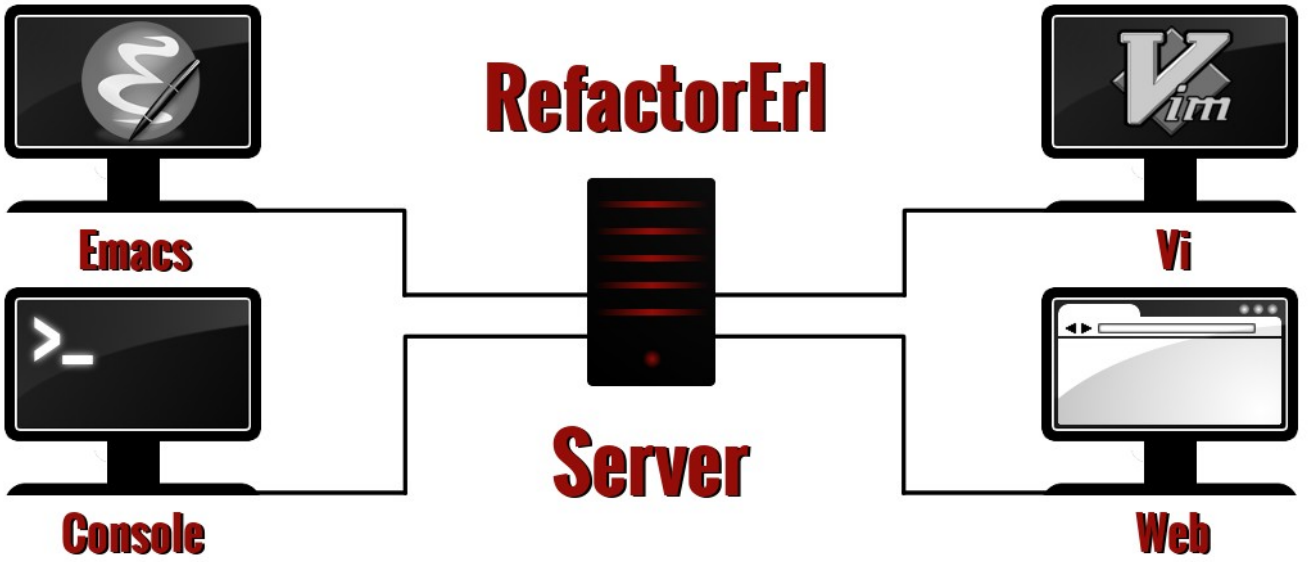
Static source code analyzer and transformer

http://refactorerl.com



Features of RefactorErl

- Semantic Program Graph
- Gather information from the code
- Find dependencies
- Investigate bugs
- Share information among team members
- Refactor the code



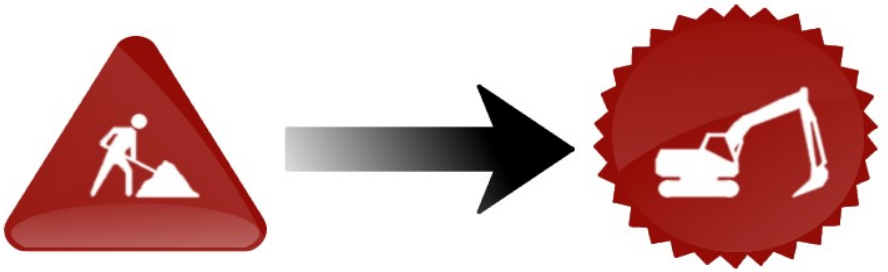
Why shall I use RefactorErl?

- Shorten time-consuming daily jobs
- Improve teamwork
- Reduce human faults
- Facilitate the deployment of releases
- Minimize the training time of newbies

Effective software maintenance

in Erlang

for Erlang



Pattern candidate discovery

- List operations
  - List comprehensions
  - Library calls (`lists:map/2`)
  - Map-like recursive functions
- Task farms and pipelines

Statistics

	RefactorErl	Wrangler	Mnesia	Dialyzer	ICE
ELOC	106k	51k	23k	17k	13k
list compr.	2647	1014	101	247	59
lists:map/2	244	93	12	2	22
lists:filter/2	131	36	7	1	7
lists:foldl	189	32	28	24	22
lists:foldr	20	1	1	0	2
recursive	1028	605	362	278	436

Map-like function

- Recursive function with list parameter `P`
- Execution paths

  Recursive:

  - returns a list `R`
  - `head(R)` does not depend on `P`, only on `head(P)`
  - `tail(R)` is from the single recursive call on `tail(P)`
  - other parameters of the recursive call are the same

  Non-recursive:

  - `P = []`
  - returns `[]`

Components

- Action performed by Worker (farm) or Stage (pipe)
- Side-effect analysis
  - Message passing
  - NIFs and global variables
  - ETS etc.
  - Process dictionary, node names
  - Exceptions
- Hygiene rather than purity

Hygienic component

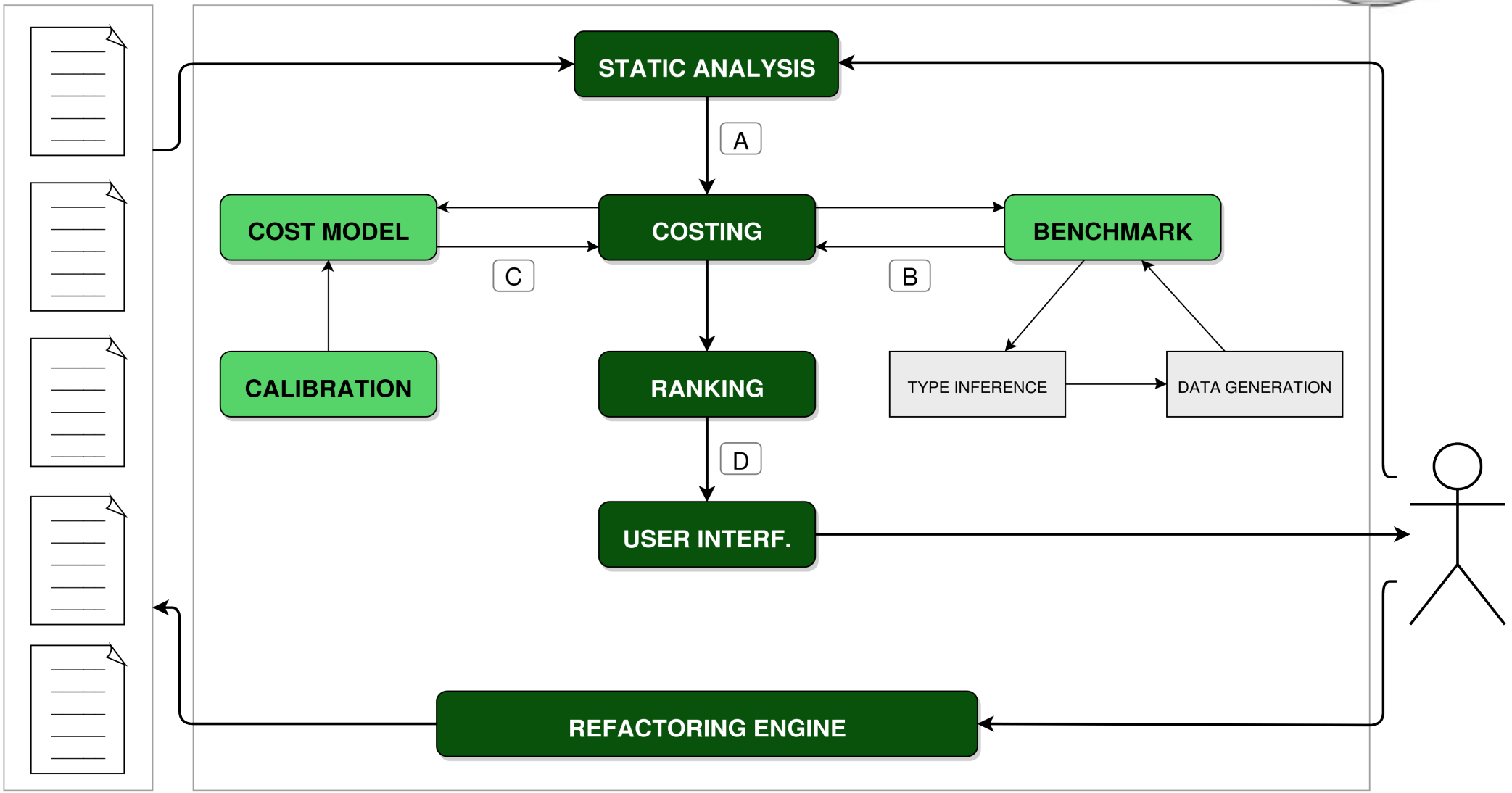
- Identify used resources
- Classify read/alter operations

  $use(C, R) \in \{ No, Read, Alter \}$

- Component set:
  components executed in parallel

  $\forall R \quad \forall C_1 \neq C_2 \in S:$
  $use(C_1, R) = Alter \rightarrow use(C_2, R) = No$

Presenting pattern candidates



Benchmarking

RefactorErl

- Split up pattern candidates into components
- Determine free variables (inputs)
- Assemble a new module
  - Components turned into functions
  - Instrumented with time measurements
- Load module
- Generate random input and profile
- Make statistics

Random input?

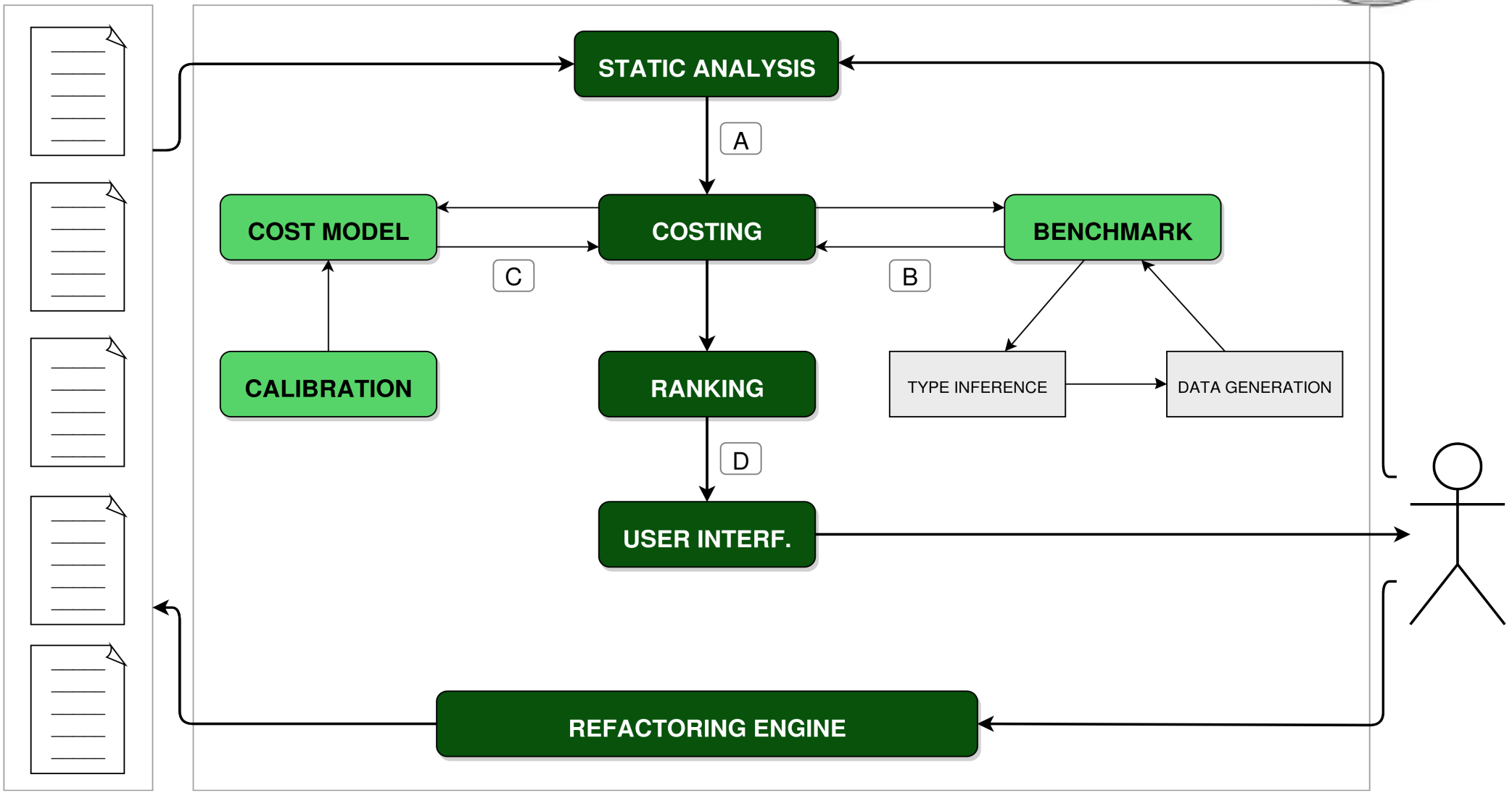
- Not always meaningful...
- ... but easy to automate!

- Find out the type of free variables
- QuickCheck generates values by type

Type inference

- Need to find a good type for free variables
  - Not the success type!
  - But describes well the possible values
- Currently we use TypEr
- Working on another approach

Presenting pattern candidates



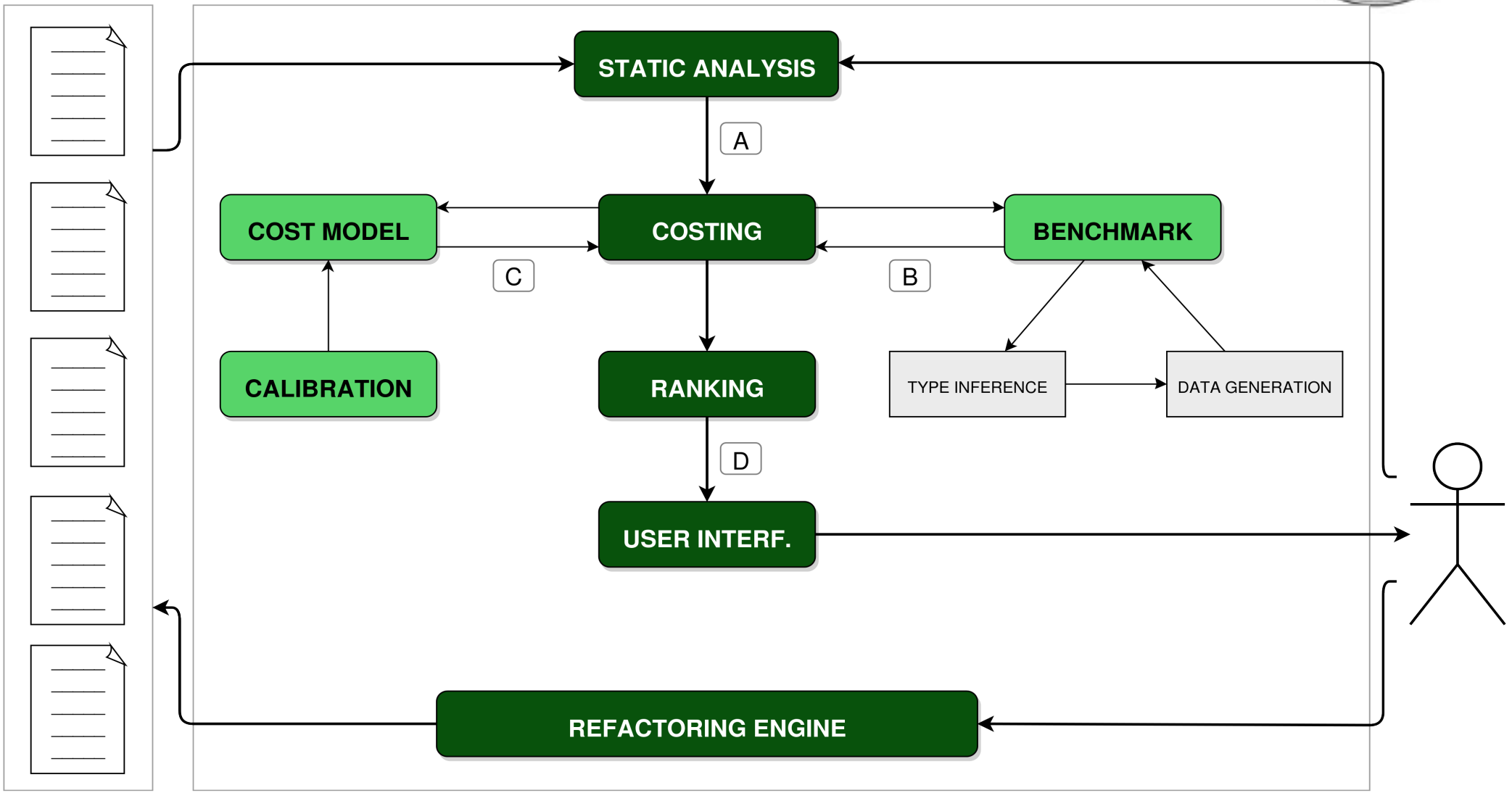
Cost model

- Approximation
- E.g. for farm:

$$T_{farm} := T_{work} * \lceil L/min(N_p, N_w) \rceil + T_{spawn} * (N_w + 2) + T_{copy}(L) * 3 + T_{spawn} + T_{copy}(L) * 2$$

- Needs calibration!

Presenting pattern candidates



Pattern Candidate Browser

- After ranking pattern candidates
- Web-based interface
  - Information for decision making
  - Not too many details
- Work on better integration with Emacs
- Services
  - Multiple users
  - Persistent results
  - Export XML, JSON, CSV, Erlang terms

Shaping transformations

- Skel is kept simple and composable
- Shaping transformations
  - Accept many syntactic forms
  - Turn them into a canonical form
  - Polishing transformations might be necessary

```
lists:filter(fun pred/1, List)

lists:append([ if pred(Item)-> [Item]; _ -> [] end
             || Item <- List ])
```

Future work

- Learn from big examples
- Support more skeletons
- Provide better ways to customize
- Add heuristics
- Improve speed
- Make the tool stable and friendly

Conclusions

- ParaPhrase Refactoring Tool for Erlang

  Wrangler + RefactorErl
- PaRTE can find parallelizable code
  - Discovers pattern candidates
  - Predicts speedup
- PaRTE offers refactorings
- PaRTE + programmer