Interactive tooling with reference attribute grammars

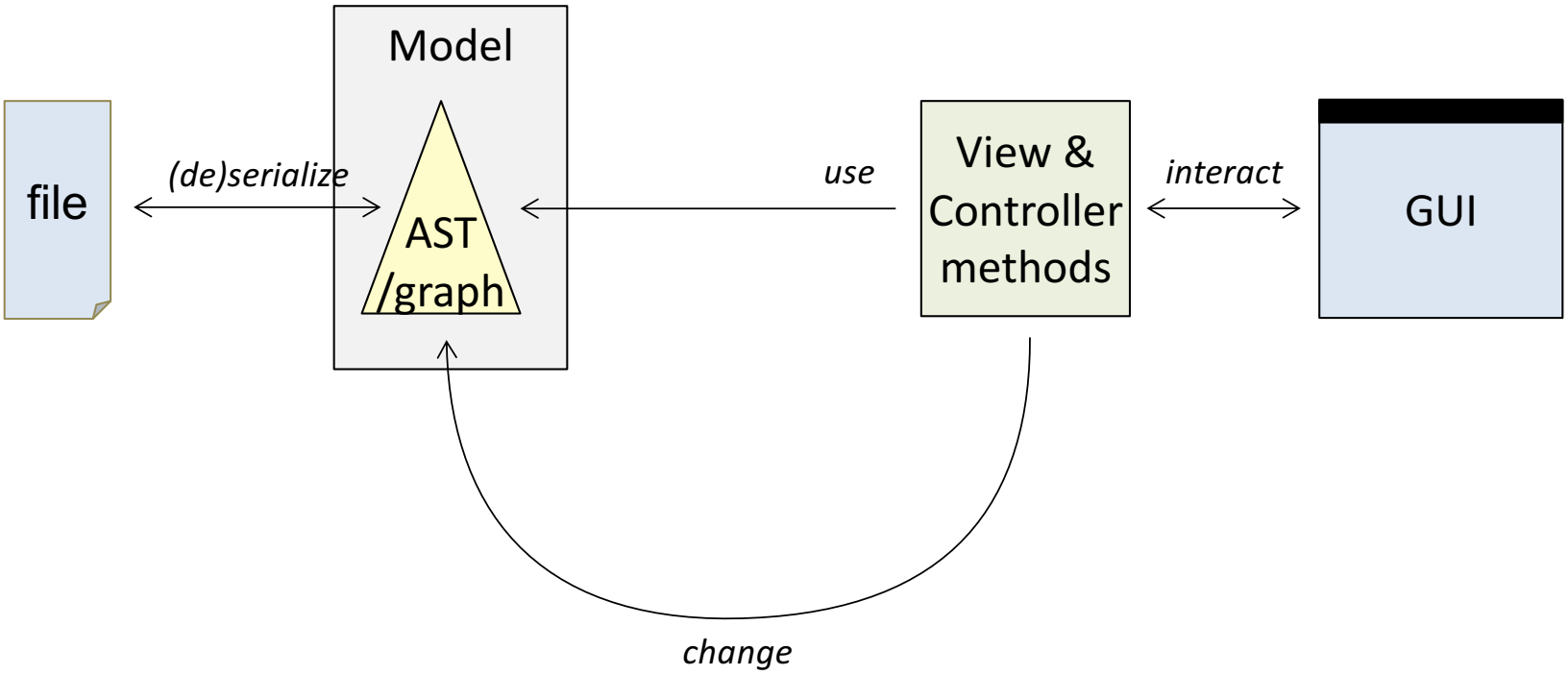
Görel Hedin

Lund University, Sweden

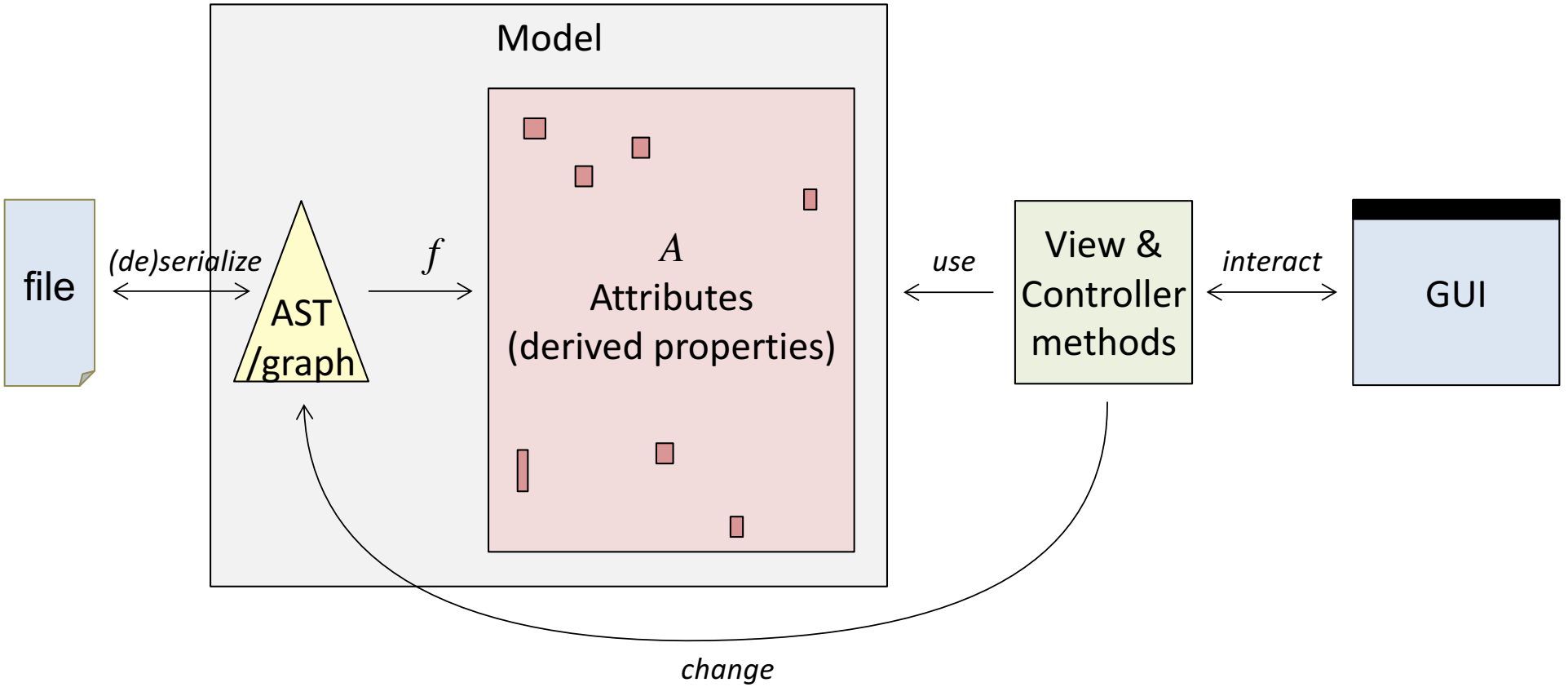
LUND UNIVERSITY

LangDev, March 8-9, 2018, Amsterdam

Basic MVC architecture



RAG MVC architecture



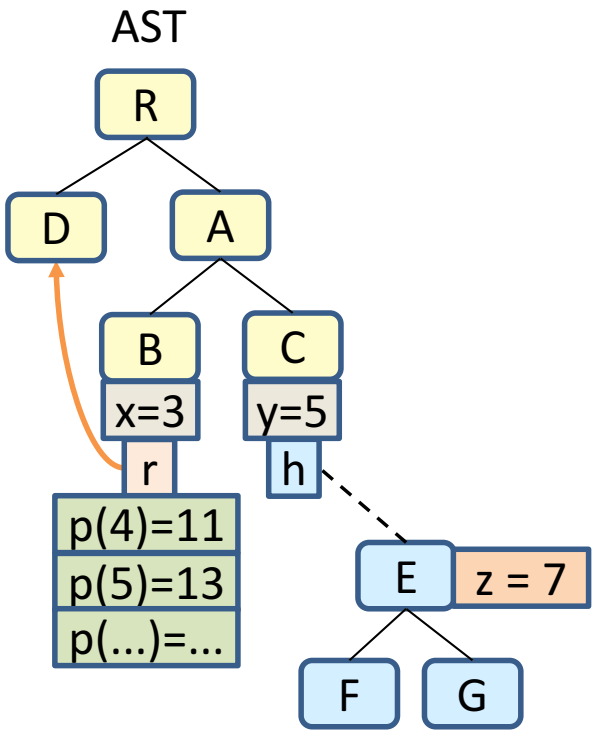
$A = f(AST)$

- rich derived information
- computed incrementally (lazy), on demand from VC
- f defined by a RAG (a set of mutually recursive equations)

RAG example

Abstract grammar

```
R ::= D A
A ::= B C
...
```



Reference Attribute Grammar

attribute: derived property of AST node
equation: definition of attribute value

synthesized: defined in node

```
syn int B.x
eq  B { x = 3 }
```

inherited: defined in parent

```
inh int C.y
eq A { C.y = B.x+2 }
```

reference: to existing AST node

```
syn D B.r
eq  B { r = ... }
```

***n**onterminal **a**ttribute* (higher-order)

```
syn nta E C.h
eq  C { h = new E(...) }
```

parameterized:

```
syn int B.p(int)
eq  B { p(i) = i*2+x }
```

attributes of nta:

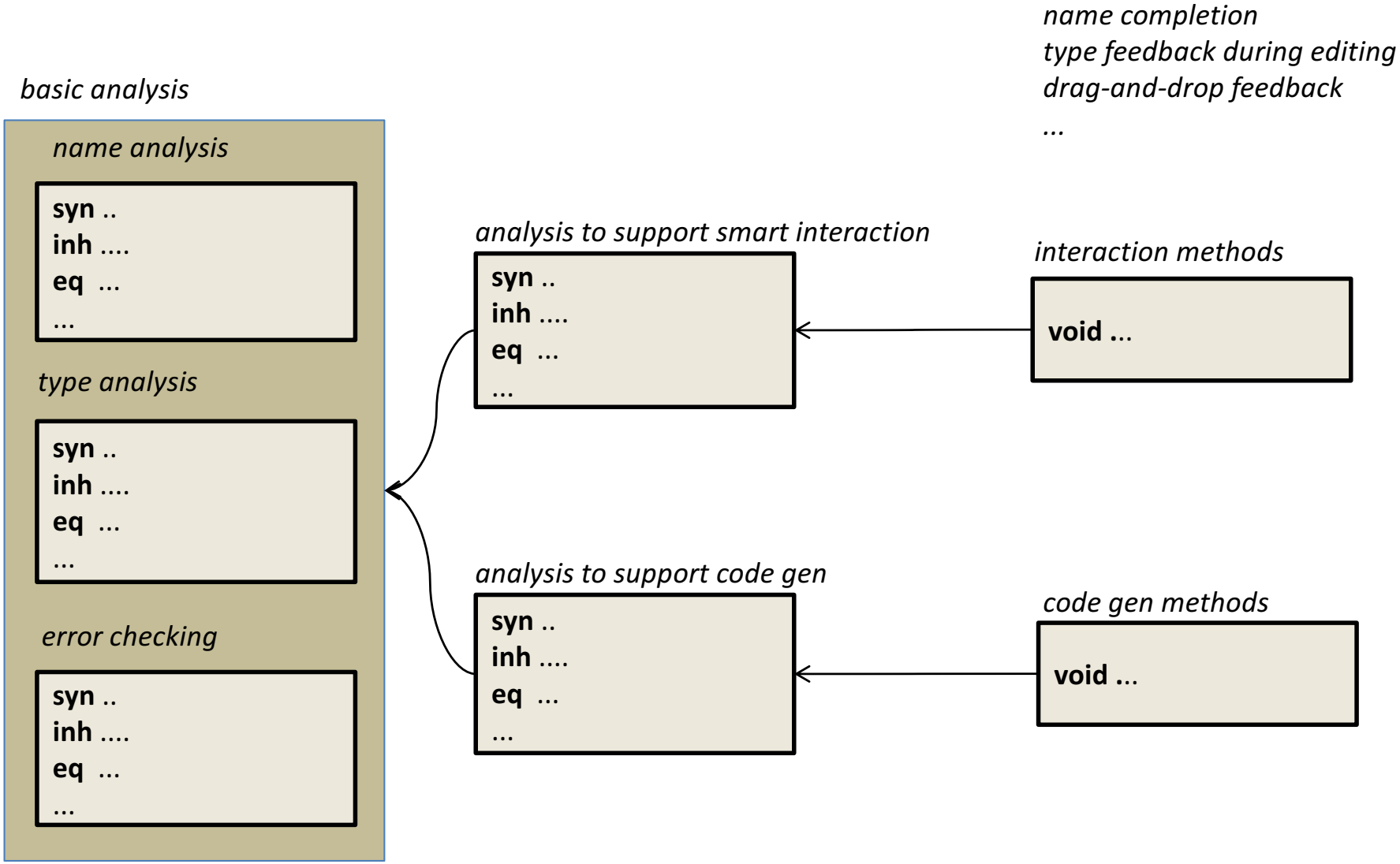
```
inh int E.z
eq  C { E.z = y+2 }
```

More: circular, collection, parameterized nta, ...

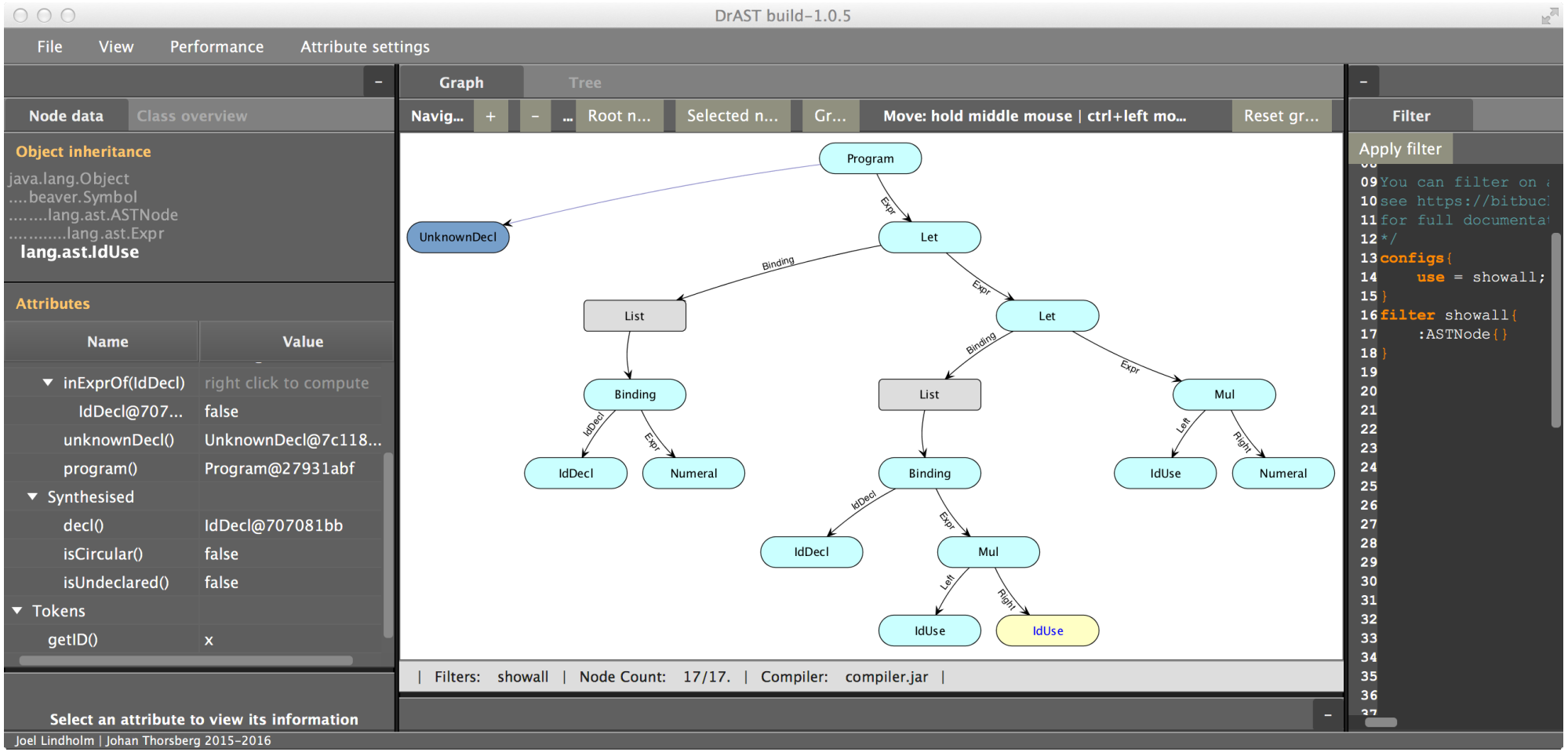
Inherently incremental: attributes not evaluated until their value is needed
Memoization: value can be cached to speed up future uses

RAG aspects

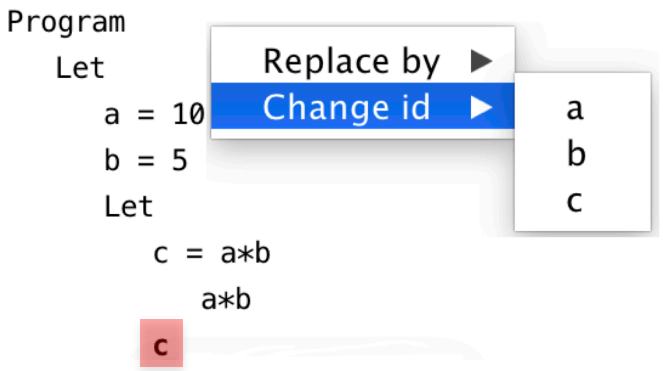
modular tool specification



DrAST – an attribute inspection tool

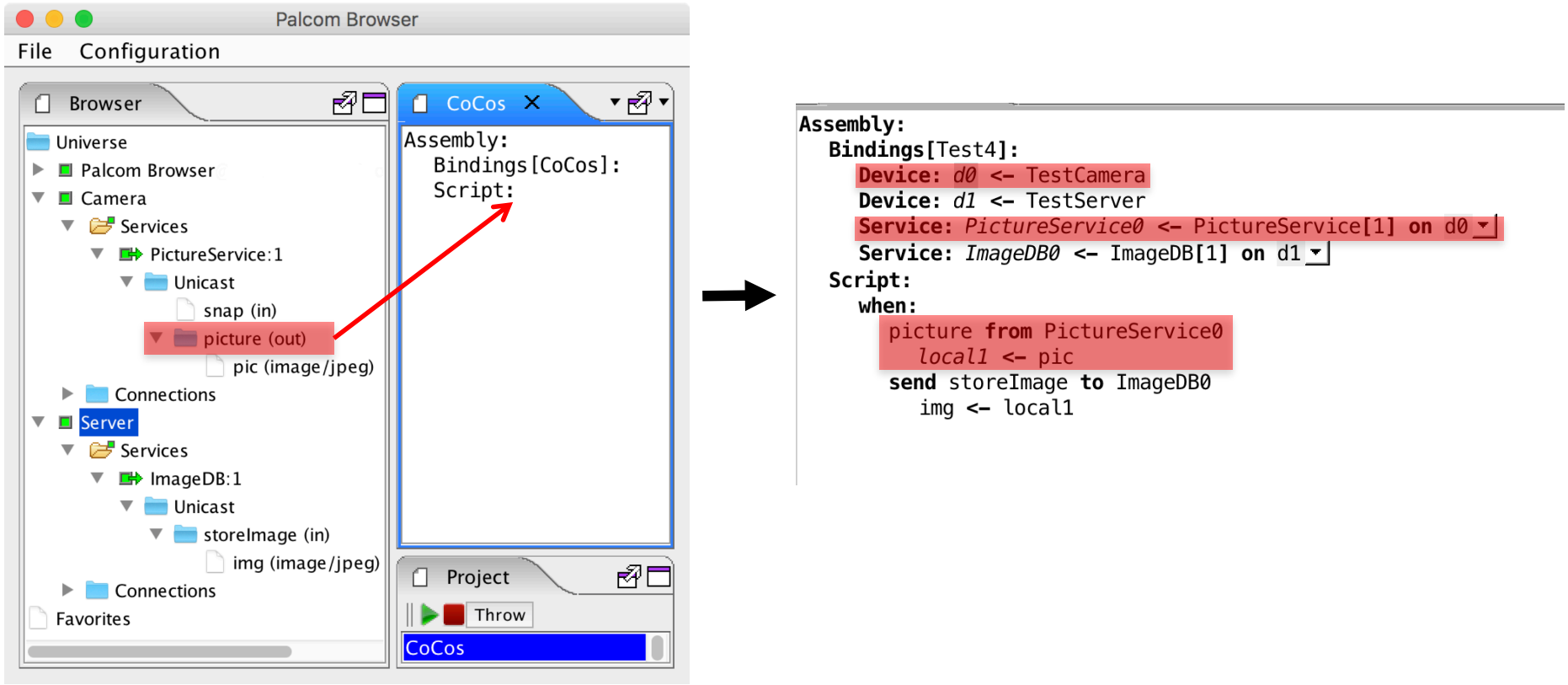


Jatte – a tunable tree editor



name completion in a Calc language

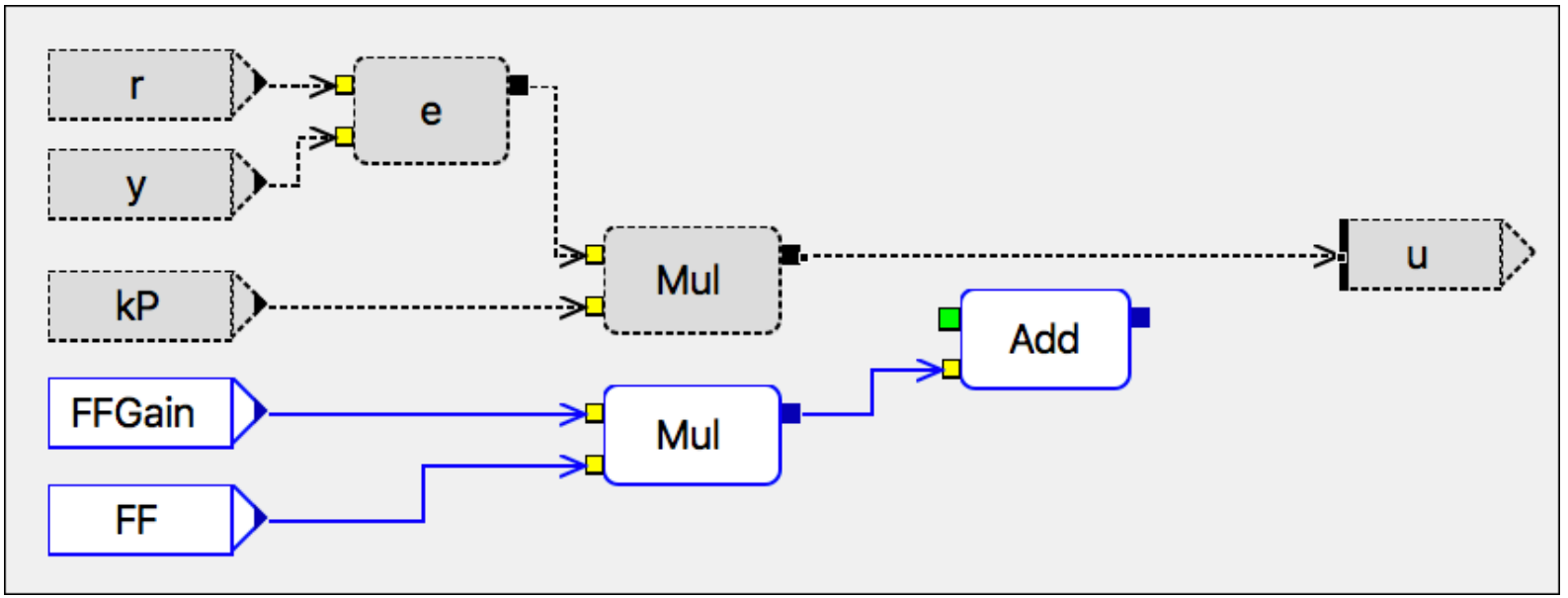
Jatte – a tunable tree editor



drag-and-drop in an IoT language

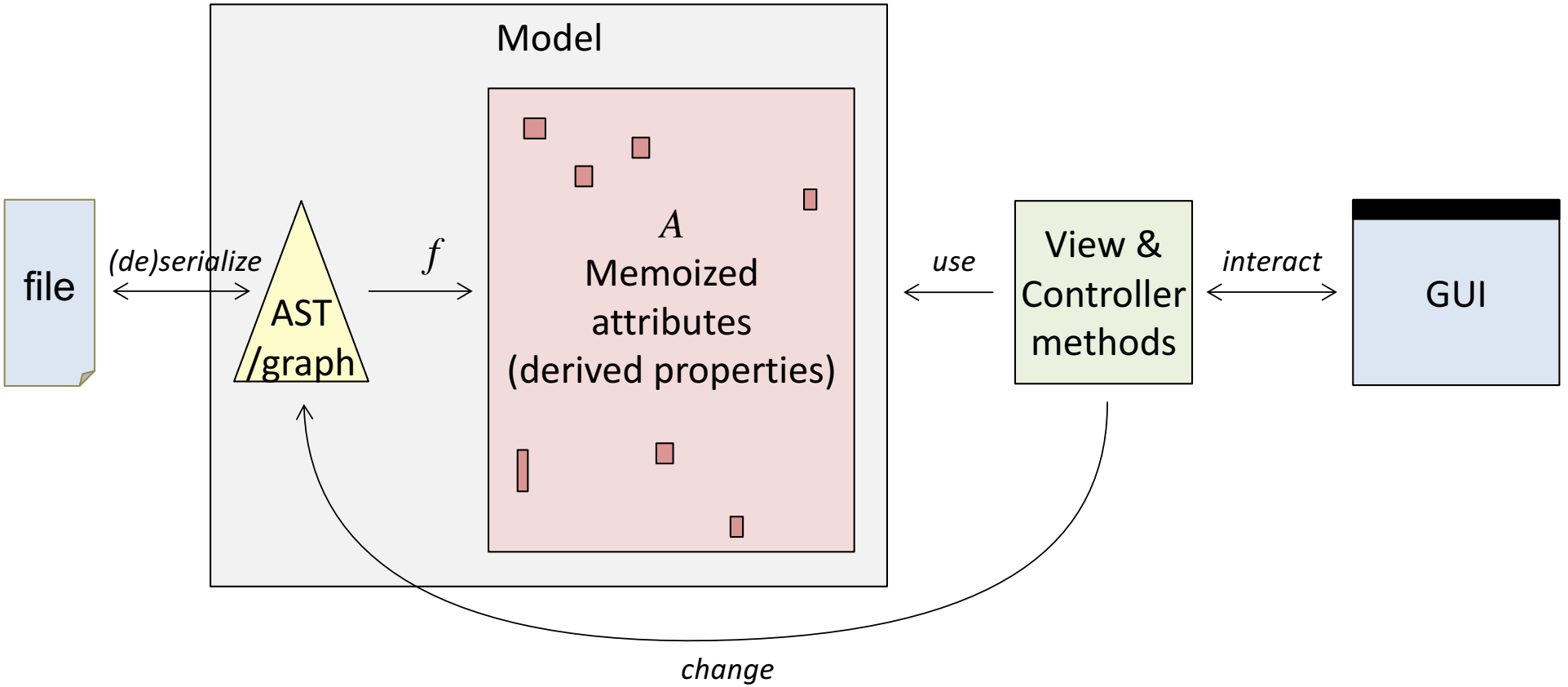
An automation language editor

(Bloqqi language)



editor suggests connectable ports based on direction and type analysis

The update problem

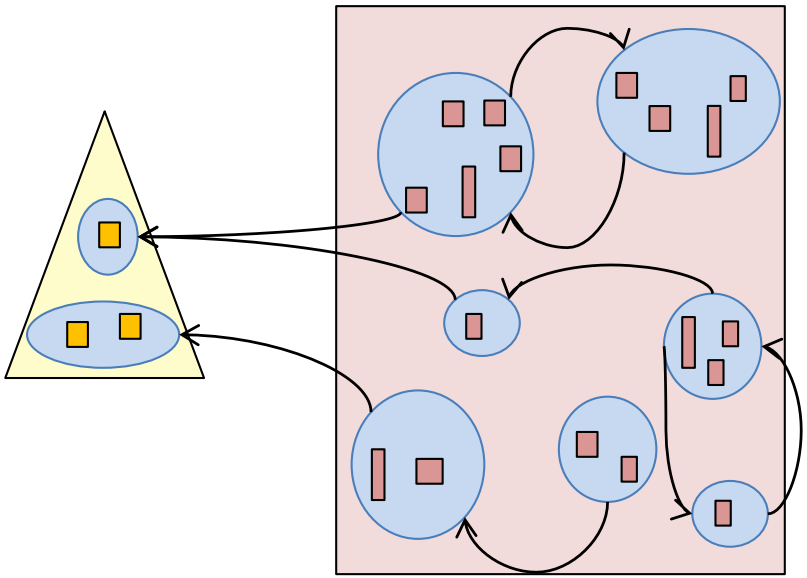


Invariant: $A = f(AST)$

What happens to memoized attributes when the AST is changed?

Note! Because of reference attributes, dependencies are not known statically.

Strategies



Attribute partitions:

- Dynamic dependency graph over partitions
- Flush dependent partitions after change

- Many partitions, higher cost, higher accuracy
- Fewer partitions, lower cost, lower accuracy

Two extremes:

- One partition per attribute instance: very costly
- One partition for all attribute instances:
  - this is what we currently use in practice
  - works surprisingly well (lazy evaluation makes re-evaluation inherently incremental)
  - but... does not scale for large programs

Ongoing work – find a middle way:

- How to partition? Subtrees, aspects, combinations, ...
- Experiment with just 2 partitions: one for edited programs, one for libraries
- Concurrent re-evaluation of flushed attributes
  - one interactive thread
  - one background thread for computing more attributes

Some references

- Jesper Öqvist, Görel Hedin: Concurrent circular reference attribute grammars. SLE 2017. https://doi.org/10.1145/3136014.3136032
- Emma Söderberg, Görel Hedin: Incremental evaluation of reference attribute grammars using dynamic dependency tracking. LU-CS-TR:2012-249. http://portal.research.lu.se/portal/files/3312049/2543179.pdf
- Alfred Åkesson, Görel Hedin: Jatte: a tunable tree editor for integrated DSLs, CoCoS 2017. https://doi.org/10.1145/3141842.3141844
- Niklas Fors, Görel Hedin: Bloqqi: modular feature-based block diagram programming, Onward 2016. https://doi.org/10.1145/2986012.2986026
- Joel Lindholm, Johan Thorsberg, Görel Hedin: DrAST: an inspection tool for attributed syntax trees (tool demo). SLE 2016. https://doi.org/10.1145/2997364.2997378