An Embedded DSL for the parameterised specification, resource analysis and scheduling of systems composed of functional building blocks

Christoph Herrmann

University of St Andrews

3 March 2010

8th IFIP WG-2.11 Meeting

Univ. St Andrews, Scotland/UK

Overview

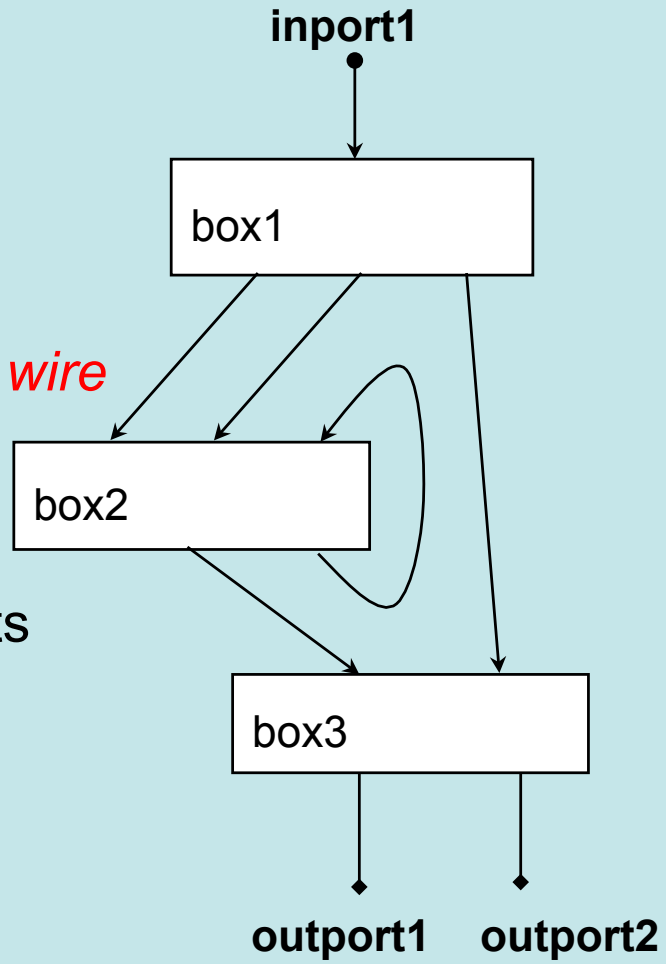
- Aims
- Motivation and background: analysis for Hume
- Difficulties, models and abstractions:
  - Functional mapping for any superstep
  - Counter automata with transitions for supersteps
  - The proposed DSL
- Abstract interpretations
  - cost analysis
  - scheduling
- Discussion

Aims

- resource analysis of compositions of functional building blocks, briefly “boxes”
- scalability: it should not matter whether there are 10 or 100000 boxes or iterations, if they are all similar
  - this is achieved by parameterisation of the design
- parameterisation of the input (loop bounds, sizes)
- take program modes into account
  - e.g. one iteration of a loop is expensive, the others are cheap

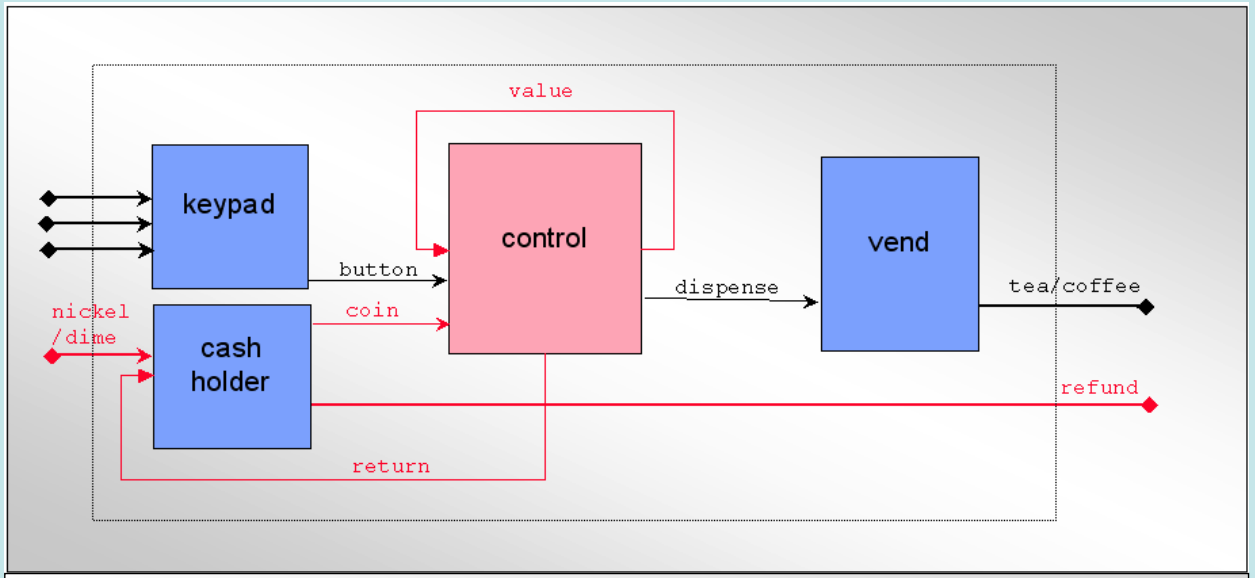
Motivation: Hume

- *boxes* structure processes
  - static process network
  - *asynchronous* communication
  - box2: automaton with *state on feedback wire*
- *functions* specify computations
  - purely functional notation
  - pattern-matching relates inputs to outputs through functional expressions



Coffee Machine

- **Hume program**
  - collection of boxes
  - wired by channels
  - example:
    vending machine



- **Hume box**
  - example:
    control box
  - pattern matching
    *: don't consume
  - pure functions
  - result values
    *: don't produce

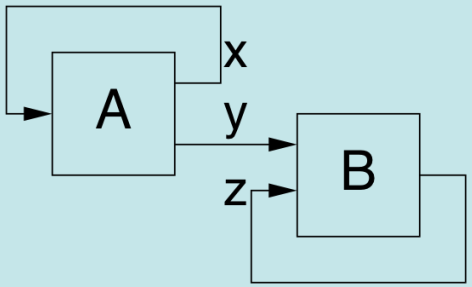
```
select drink cost v = if v >= cost then (drink, v-cost, *)
                                   else ( *, v, * );

box control
in  ( coin :: Coins, button :: Buttons, value :: Int )
out ( dispense :: Drinks, value' :: Int, return :: Int )
match
    ( Nickel, *,       v )  ->  ( *, v + 5, * )
  | ( Dime,   *,       v )  ->  ( *, v + 10, * )
  | ( *,      BCoffee, v )  ->  select Coffee 10 v
  | ( *,      BTea,    v )  ->  select Tea 5 v
  | ( *,      BCancel, v )  ->  ( *, 0, v )
  ;
```

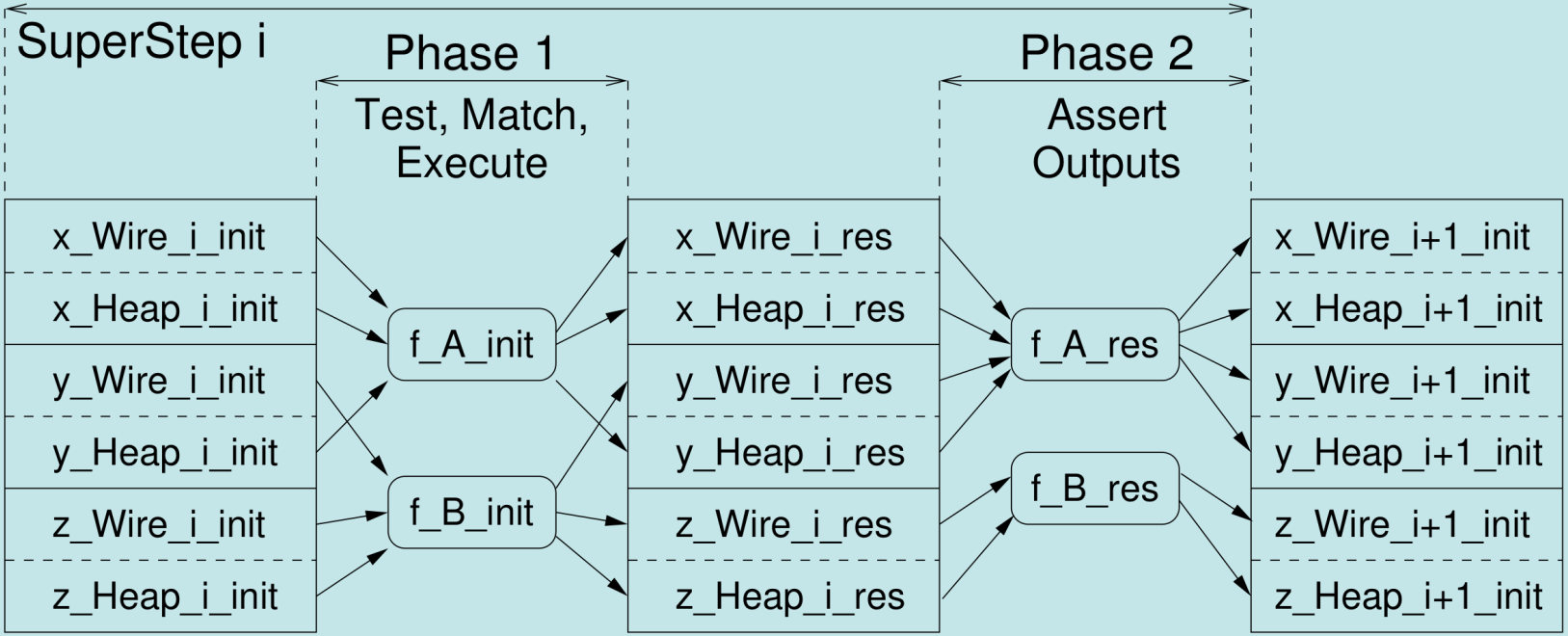
1. Abstraction: Box Function

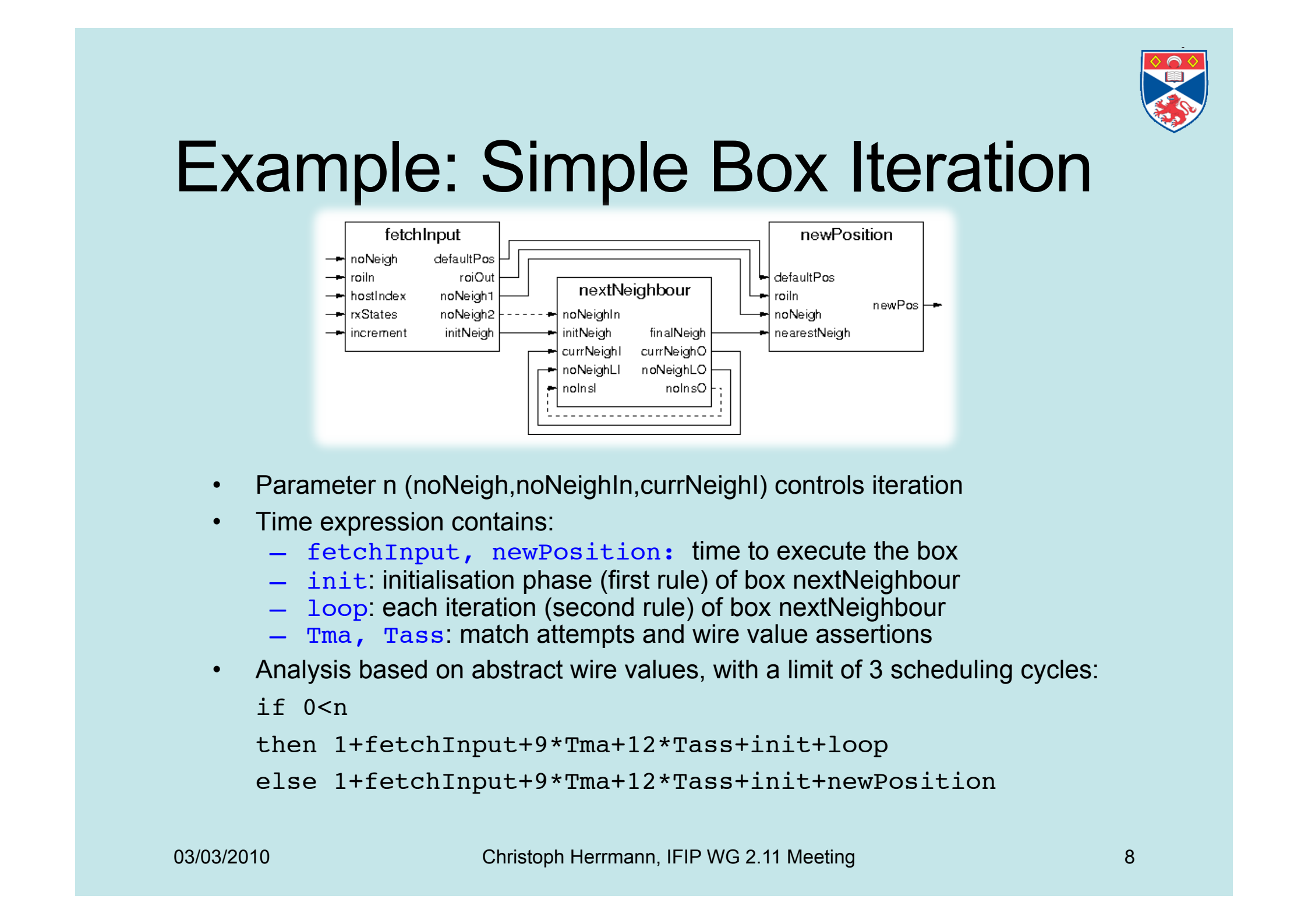
- Reduced box behaviour
  - iteration counters, size information, etc.
- Specify operational conditions
  - box runnable? which operands consumed?
- Specify extended abstract box function
  - determine which pattern matched
  - decrement of iteration counters, etc.
  - cost increment

2. Abstraction: Coordination



- Mapping between wire/heap vectors
- Covers runnability and blocking





Example: Simple Box Iteration

- Parameter n (noNeigh,noNeighIn,currNeighI) controls iteration
- Time expression contains:
  - `fetchInput, newPosition:` time to execute the box
  - `init`: initialisation phase (first rule) of box nextNeighbour
  - `loop`: each iteration (second rule) of box nextNeighbour
  - `Tma, Tass`: match attempts and wire value assertions
- Analysis based on abstract wire values, with a limit of 3 scheduling cycles:

```
if 0<n
then 1+fetchInput+9*Tma+12*Tass+init+loop
else 1+fetchInput+9*Tma+12*Tass+init+newPosition
```

Difficulties

- Current capabilities
  - model Hume at box coordination level
    - using sum and product types in symbolic expressions
  - simplification in the absence of cycles with unknown number of iterations
- Challenges
  - nested loops expressed at the coordination level
    - tackling at the design level: e.g. loop box template
  - unknown parameters controlling iteration
    - complicated simplification, limits of decidability

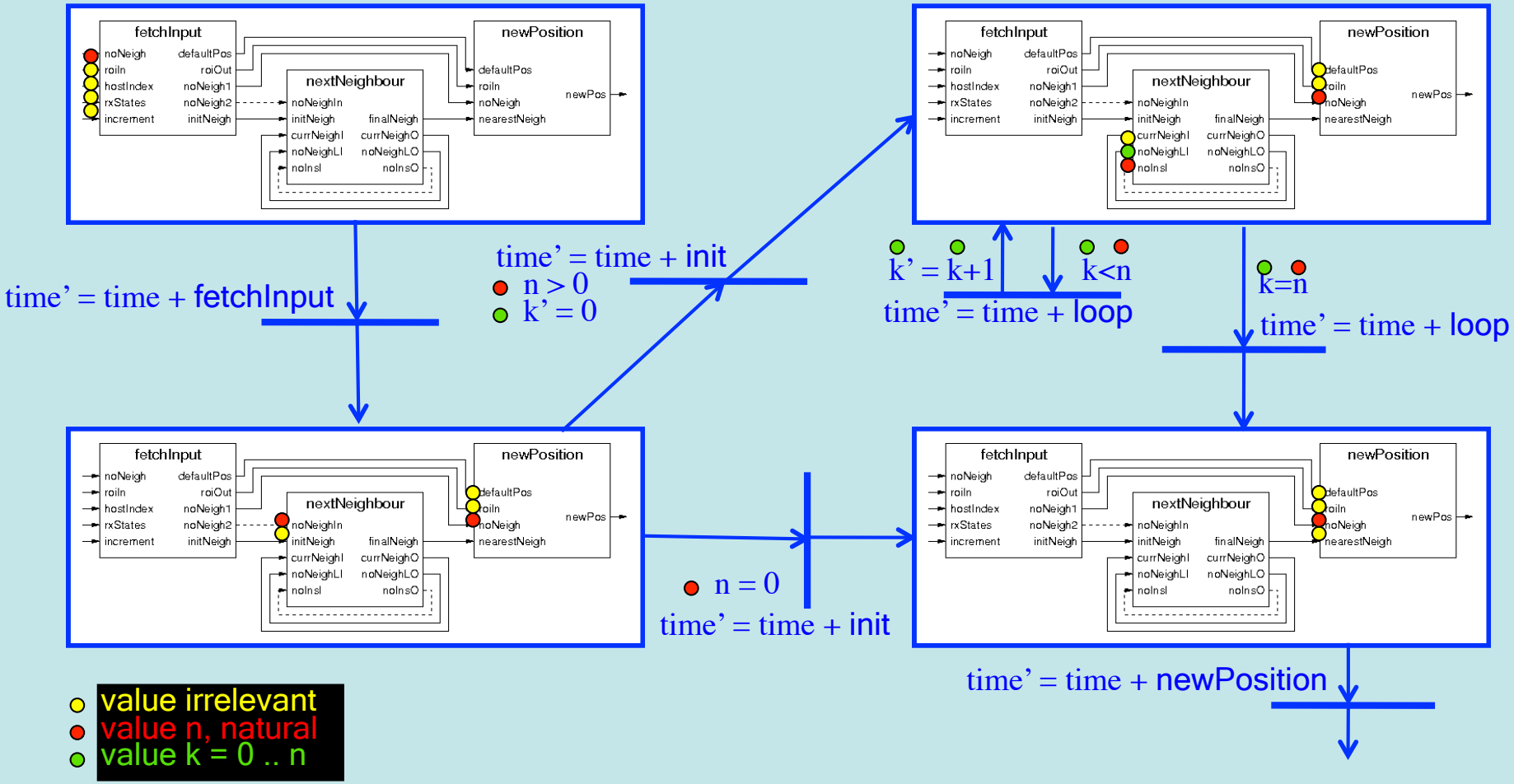
Tackling Analysis Complexity

- 3. Abstraction: translation into counter automata -

- Translation from the box execution model
  - state ≙ set of values of wire vectors
  - transition ≙ superstep (set of executions)
- Identify variables relevant for the analysis
  - control parameters and iteration counters
  - resource counts
- Cycles of state transitions
  - cluster cycles, parameterise cluster by iterations
  - express transition function inductively
  - solve recurrence (e.g., interpolate & verify)

Counter Automata

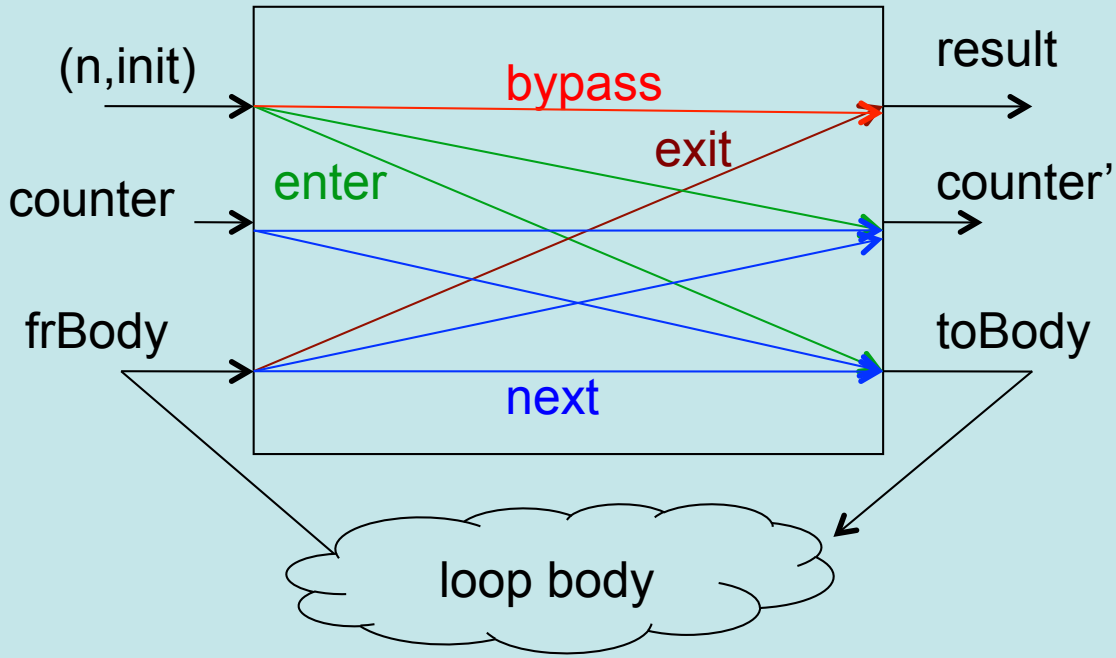
analysis now: ignore what's in the boxes!



Treatment of Cyclic Executions

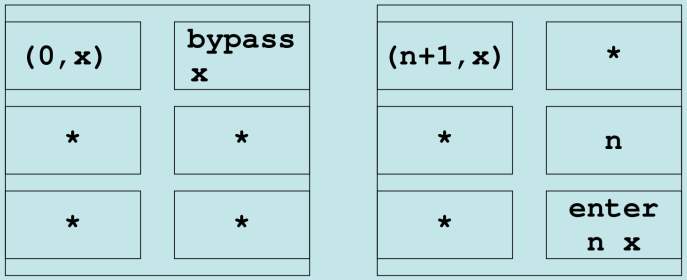
(Example Loop Box)

bypass (counter=0): ((0,init), * , *) → (bypass init, *, *)
enter (counter>0): ((n+1,init), * , *) → (* , n, enter n init)
next (counter>0): (* , n+1, frBody) → (* , n, next n frBody)
exit (counter=0): (* , 0, frBody) → (exit frBody, *, *)



Inductive Analysis for Loop Box

(4 state transitions)



$$t(n) = \begin{cases} t_{bypass} & \text{if } n = 0 \\ t_{enter} + (n-1)\cdot t_{next} + t_{exit} + \sum_{i=0}^{n-1} t_{body}(i) & \text{otherwise} \end{cases}$$

Further Difficulty

- Problem: identify cycles in the automaton controlled by iteration counters
- Subproblem: decide predicates occurring in symbolic expressions
- Ad-hoc solution: let the designer of the system provide that information

4. Abstraction: Embedded DSL

- Host Language: Haskell
- Language Features
  - cost for execution of a rule of a box
  - fixed parallel/sequential composition of boxes
  - selection depending on an expression
  - indexed sequential/parallel aggregations

Syntax of the DSL

```
M : language of box compositions
e : language of symbolic expressions

data M e
 = M String e                    -- box/rule with cost in e
 | Seq [M e]                     -- fixed sequence
 | Par [M e]                     -- fixed independent set
 | Alt (e -> Int) e [M e]        -- choice
 | MS e (e -> M e)               -- sequential aggregation
 | MP e (e -> M e)               -- independent aggregation
```

Syntax of Expressions

(one potential instantiation of sizes and indices)

```
data Exp = C Int                    -- integer constant
         | V String                 -- variable
         | Exp :+: Exp              -- addition
         | Exp :^: Exp              -- maximum
         | Sum Exp String Exp       -- sum number var element
         | Max Exp String Exp       -- max number var element
```

Choice

```
Alt
   (e -> Int)   -- decision procedure provided by designer
   e            -- index of the choice
   [M e]        -- list of subdesigns for each choice
```

Note:

- choices (case distinctions) are of little value inside cost expressions
- moving case distinctions out of compositions leads to exponential explosions, range partitioning not always possible (e.g., n mod 2)
- general decision procedure for e does not exist
- ad-hoc solution: let the designer provide the decision procedure
  - specification within a type class could not regard context
  - therefore: provide procedure as an argument to the constructor

Sequential Aggregation

```
MS
   e              -- size
   (e -> M e)     -- maps index to subdesign
```

Note:

- simplification of size not required immediately
- resource calculation results in a symbolic sum expression
- in simple cases this just increments the degree of a polynomial, e.g.,

$$time(\text{MS } n\ (\lambda i \rightarrow M\, i)) = \text{Sum } n\ (\lambda i \rightarrow i) = \sum_{i=0}^{n-1} i = 1/2n^2 - 1/2n$$

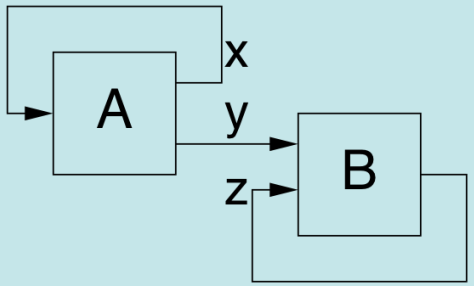
Timing Calculation

```
class Sys e where
 calcT :: M e -> e

instance Sys Exp where
 calcT (M _ n) = n
 calcT (S xs) = foldl (:+:) (C 0) (map calcT xs)
 calcT (P xs) = foldl (:^:) (C 0) (map calcT xs)
 calcT (Alt p n xs) = calcT (xs !! (p n))
 calcT (MS n f) = let v = fresh (f (V ""))
                  in Sum n v (calcT (f (V v)))
 calcT (MP n f) = let v = fresh (f (V ""))
                  in Max n v (calcT (f (V v)))
```

Ex.: Factorial Computation

```
Boxes/rules: A0: 0      -> (*,*)
             A1: n      -> (n-1,n)
             B0: (n,f)  -> n*f
             B1: (n,*)  -> n
```




```
Input: let m x = M x (V x)
       in Seq [m "A1",
               Par [m "A1", m "B1"],
               MS ((V "n"):+:(C (-2)))
                  (\x -> Par [m "A1", m "B0"]),
               Par [m "A0", m "B0"],
               m "B0"]
```

Timing Calculation

- Input: DSL program for fac(n), n>1
- Output: time(n)

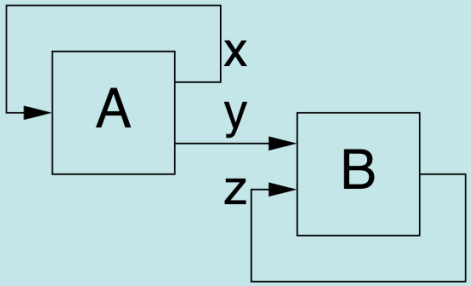
```
Input: let m x = M x (V x)
       in Seq [m "A1”,
               Par [m "A1", m "B1"],
               MS ((V "n"):+:(C (-2)))
                  (\x -> Par [m "A1", m "B0"]),
               Par [m "A0", m "B0"], m "B0"]
Output: (((((0+A1)+((0^A1)^B1))
         +(SUM x0<(n+-2):((0^A1)^B0)))
         +((0^A0)^B0))+B0)
simplified: A1+(A1^B1)+(n-2)*(A1^B0)+(A0^B0)+B0
```

Scheduling

- Output: set of boxes/rules for each superstep

```
Input: let m x = M x (V x)
       in Seq [m "A1",
               Par [m "A1", m "B1"],
               MS ((V "n"):+:(C (-2)))
                  (\x -> Par [m "A1", m "B0"]),
               Par [m "A0", m "B0"],
               m "B0"]
```

Output:
- step 0 = $\{A_1\}$
- step 1 = $\{A_1, B_1\}$
- step i | i<n = $\{A_1, B_0\}$
- step i | i==n = $\{A_0, B_0\}$
- step i | i==n+1 = $\{B_0\}$



Summary of Abstractions

1. From function for each rule of each box
   - already analysable by another tool
2. From runnability and blocking
   - no expressive power, only overhead for the analysis
   - replace by mapping between vectors of wire values
3. From wire availability and value
   - equivalent sets become state of a counter automaton
   - iteration counters taken over to the automaton
4. From iteration within cycles in the automaton
   - express cycles by a parameterised sequence

Discussion