UNIVERSITY OF Southampton

Verification patterns for Refinement

Michael Butler

22 March 2016

BCTCS'16, Belfast

Constructing correct programs

- **Analysis** – much effort devoted to verifying programs are correct
  - but, often programs are not correct
- **Synthesis** - but how do we construct programs such that they are correct?

- Transformational approach to synthesis
  - apply correctness preserving transformations to specs
  - e.g., Refinement Calculus (sequential programs), Circus (sequential + distributed)
- Posit and prove approach to synthesis
  - construct models at different levels of abstraction and prove refinement (Z, VDM, B, Event-B,…)

Event-B

- Event-B model:
  - Variables: integers, abstract sets, power sets, relations, functions, …
  - Invariants
  - Events: guarded atomic actions

- Refinements: Verify conformance between models using incremental approach

- Rodin: open source toolset for modelling, verification and simulation

- Industrial users

- Originator: Jean-Raymond Abrial

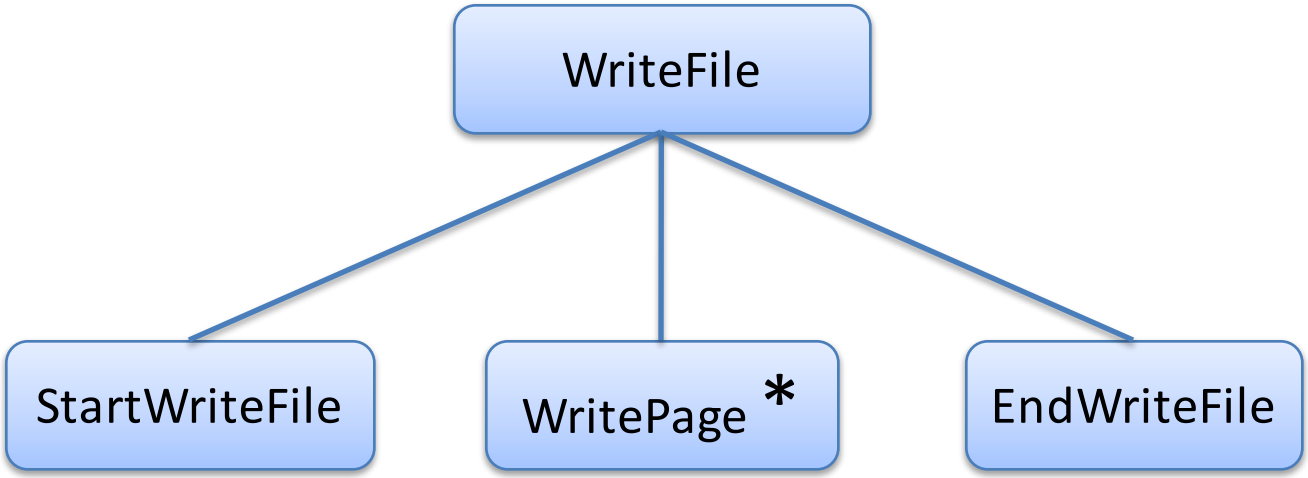
Event-B refinement

- Refinement (posit and prove)
  - one-to-many event refinement
  - new events (stuttering steps)
  - Proof method: gluing invariants, proof obligations
- Flexible: allows complex relationships between abstract and refined models
  - Decomposing large atomic steps to more fine-grained steps – concurrency, distribution easily modelled
- But (perhaps) too much flexibility
  - lacks some of the algorithmic structure provided by refinement calculus

This talk

- Support transformational style as verification patterns in Event-B
  - Patterns determined by specification structure
  - ERS (Event Refinement Structures):
    - Structured graphical representation
- But also retain advantages of Event-B style refinement:
  - separation of concerns through layered refinement (eases the verification burden)
  - easy modelling of concurrency / distribution

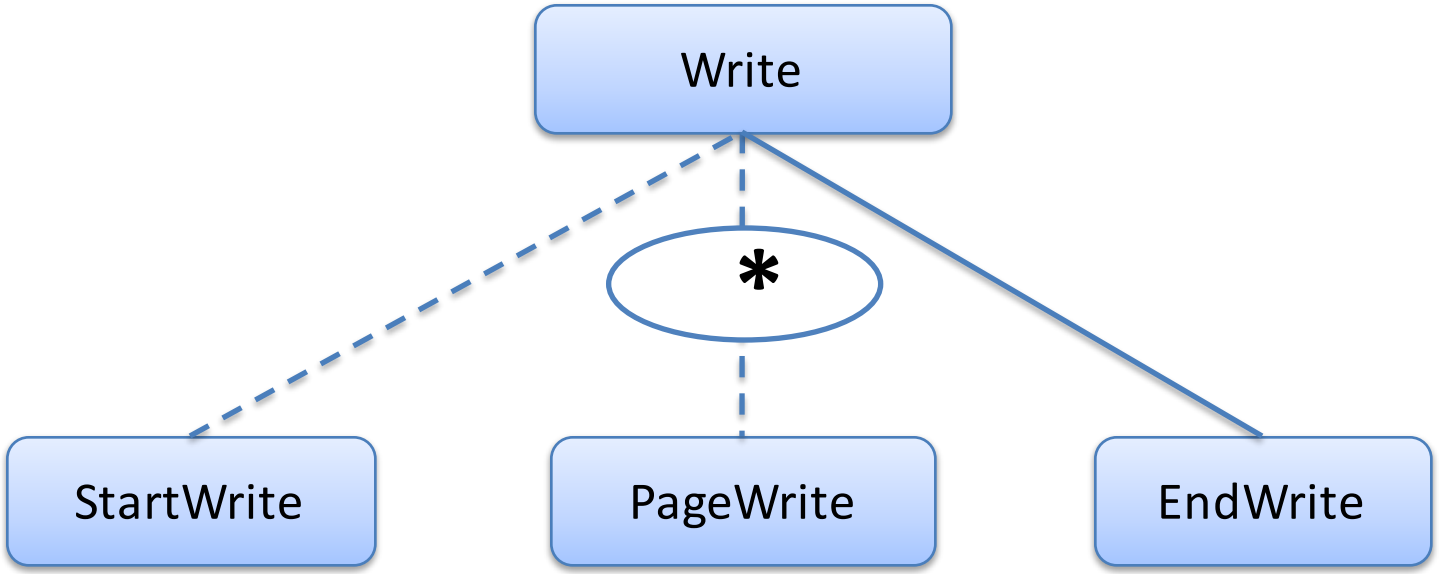
WriteFile sequencing as a Jackson Structure Diagram



Sequencing is from left to right

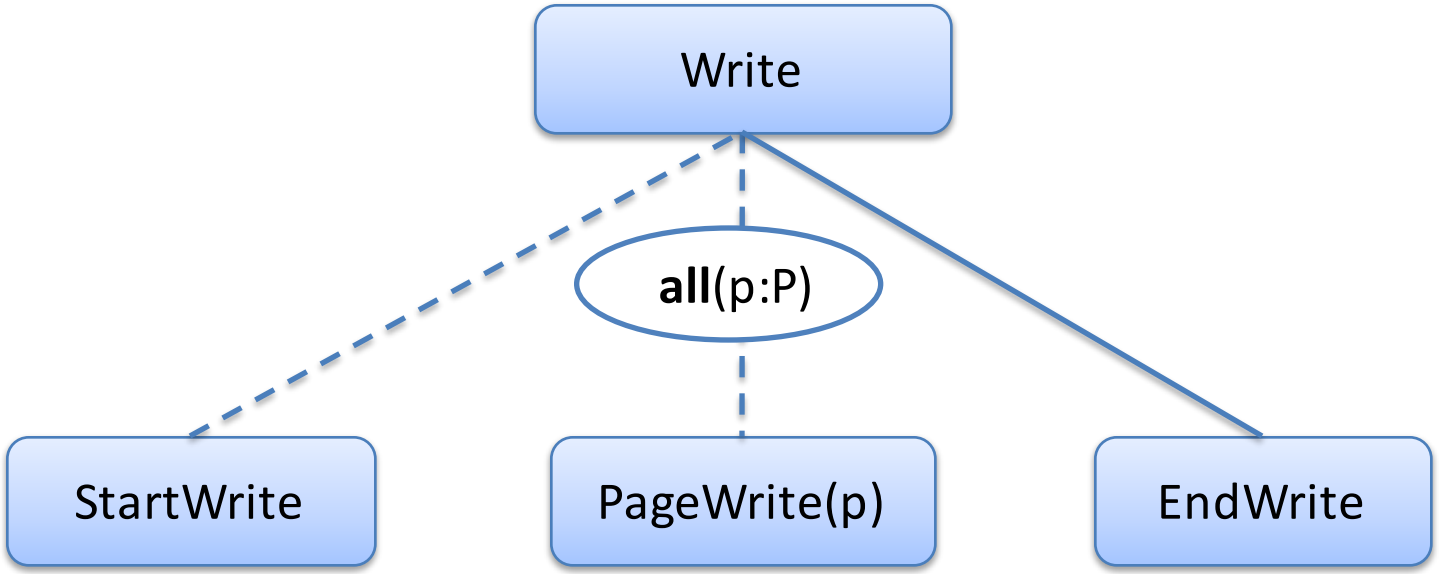
* signifies iteration

Adapting the diagrams



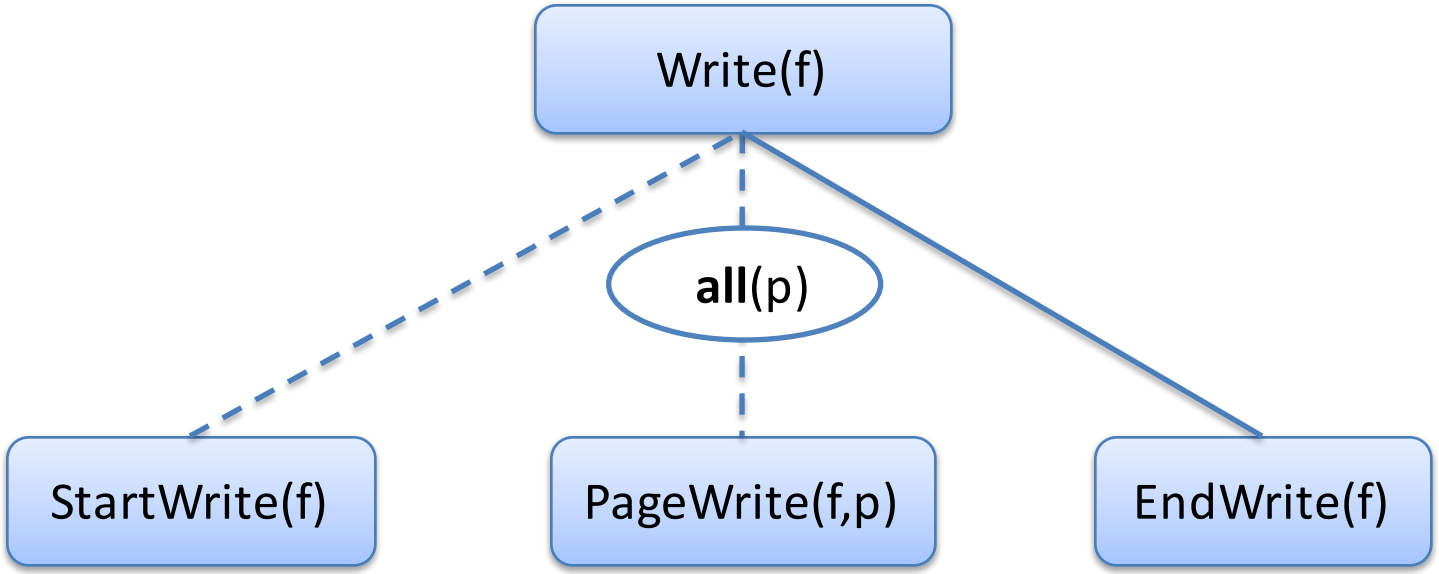
- Attach the iterator to an arc rather than a node to clarify atomicity
- Events are represented by leaves of the tree
- Solid line indicates *EndWrite* refines *Write*
- Dashed line indicates new events refining *skip*

Nondeterministic forall



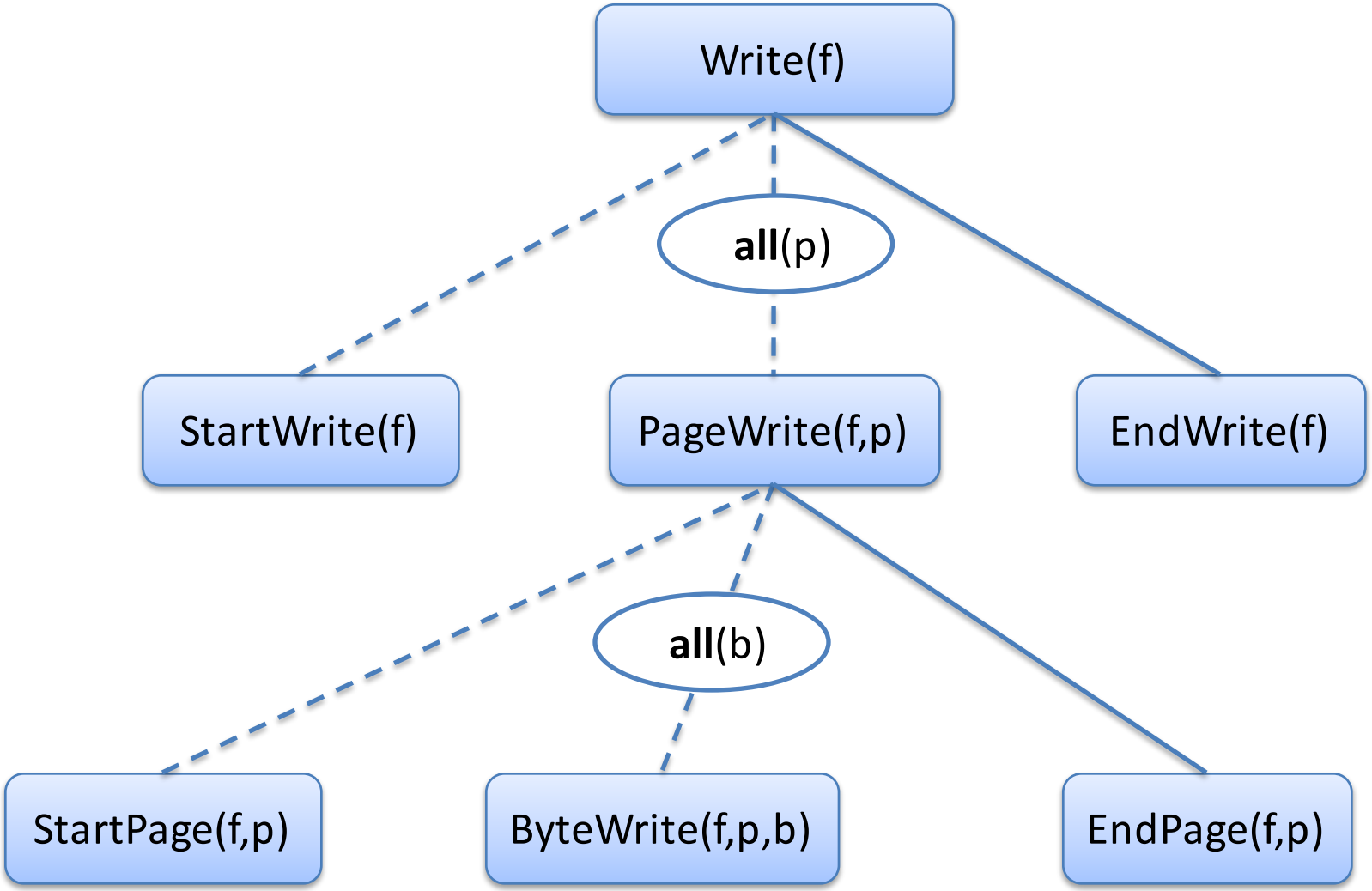
- pages may be written after *StartWrite* has occurred
- the writing is complete (*EndWrite*) once **all** pages have been written
- order of *PageWrite* events is nondeterministic

Interleaving of multiple instances



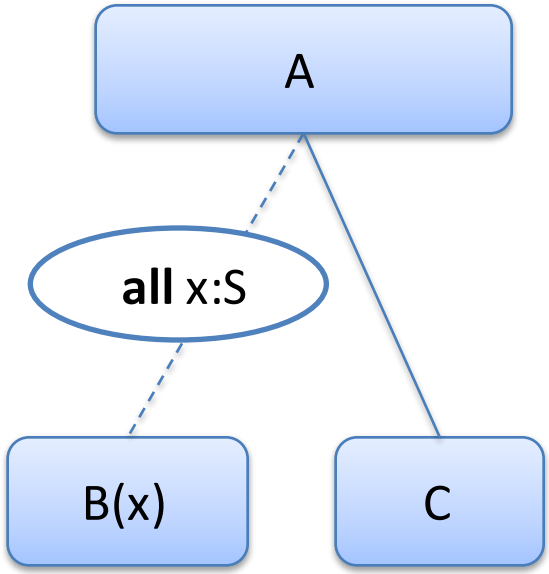
- Multiple write "processes" for different files may interleave
  - (sub-)events of *Write(f1)* may interleave with (sub-)events of *Write(f2)*
  - (sub-)events of *Write(f1)* may interleave with (sub-)events of *Read(f1)*
- *interleaving can be reduced with explicit guards (e.g., write in page order)*

Hierarchical refinement

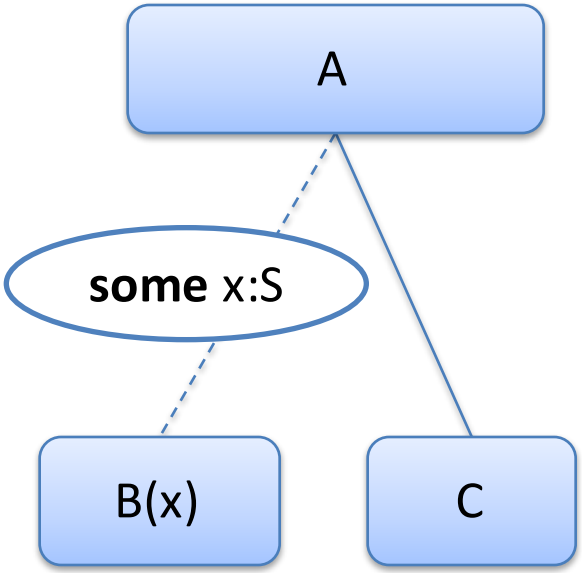


All and *Some* constructs

- **All**: C may occur once B has occurred for *all* x in S

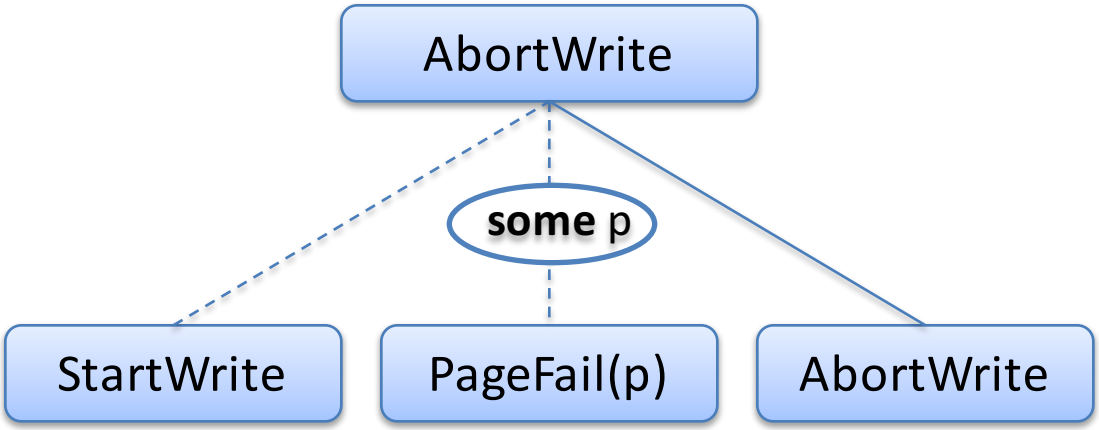


- **Some**: C may occur once B has occurred for *some* x in S



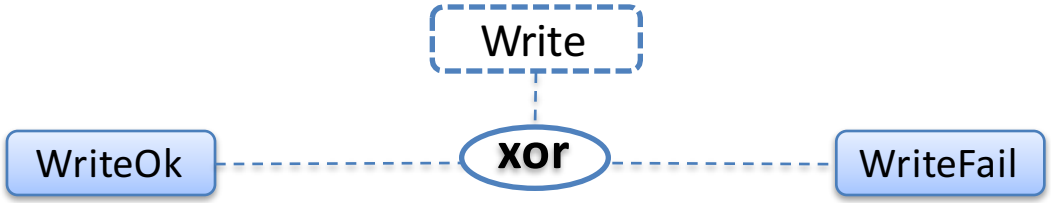
- NB: B(x') may occur after C

Treating failure in file write



- *AbortWrite* may occur if *PageFail(p)* occurs for some page *p*
- Weak: *PageFail(p')* may occur for other *p'* after *AbortWrite*

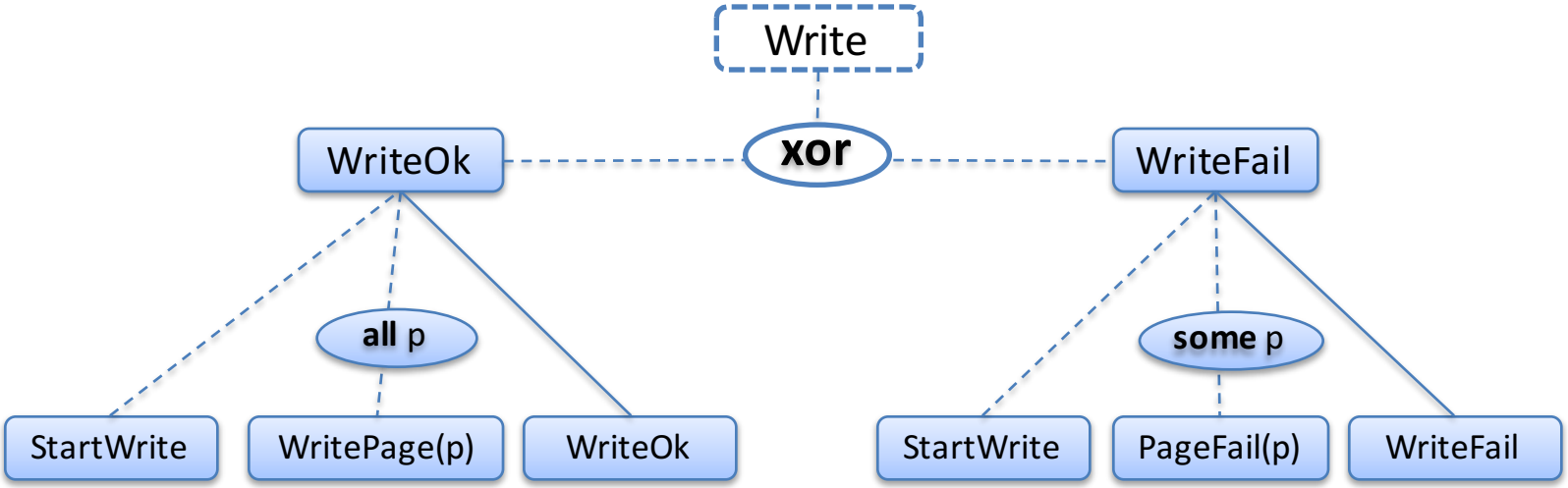
Separation of concerns



WriteOk ≙
begin
disk := file
end

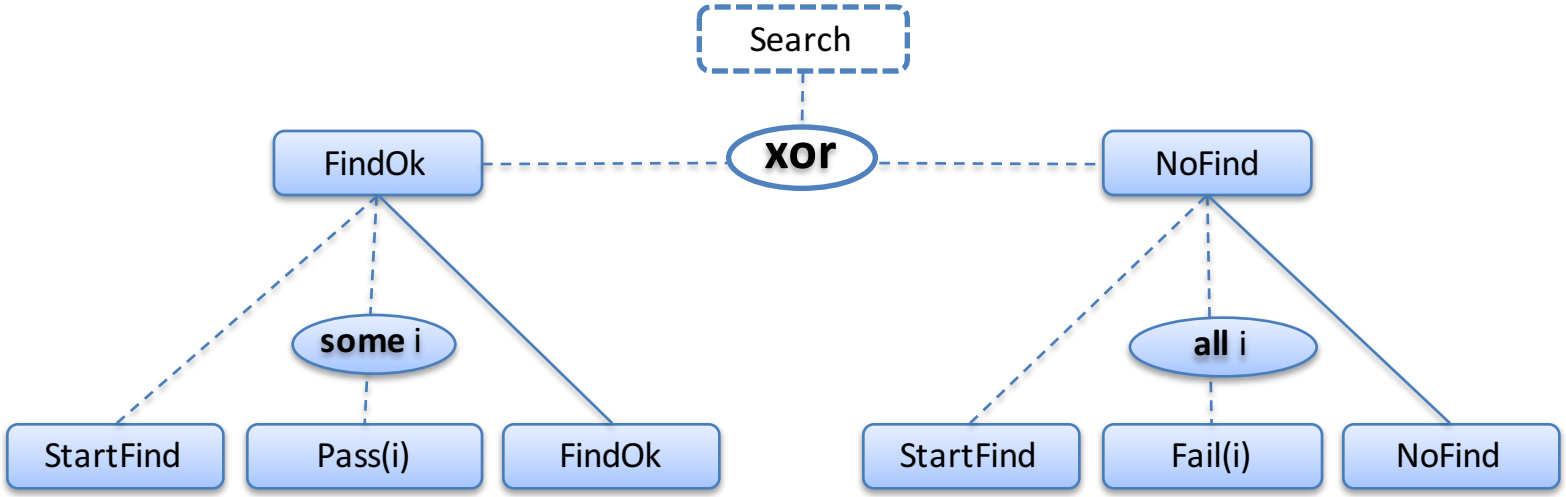
WriteFail≙
begin
skip
end

Layered refinement



- M0: two events - *WriteOk* and *WriteFail*
- M1: refine atomicity of *WriteOk*
- M2: refine atomicity of *WriteFail*

Search



- *FindOk*: $\exists x \in S \bullet P(x)$

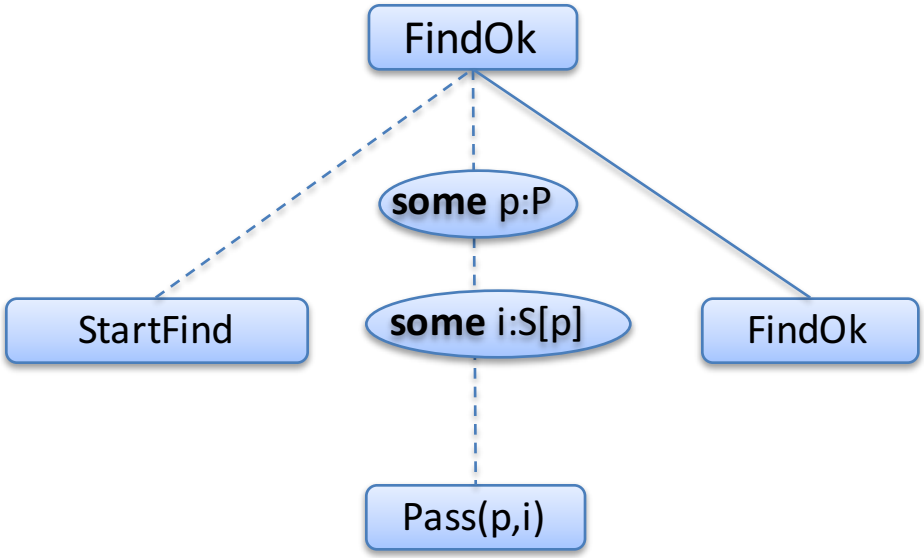
or

- *NoFind*: $\forall x \in S \bullet \neg P(x)$

Merge to form a (less abstract) sequential model

```
StartFind ;
for i in S do
    Fail(i)
    []
    Pass(i) ; exit
od ;
if exit then FindOk else NoFind fi
```

Alternatively refine to parallel model



- Partition S so that search is farmed out to multiple processors $p \in P$
- This is a simple refinement step in Event-B

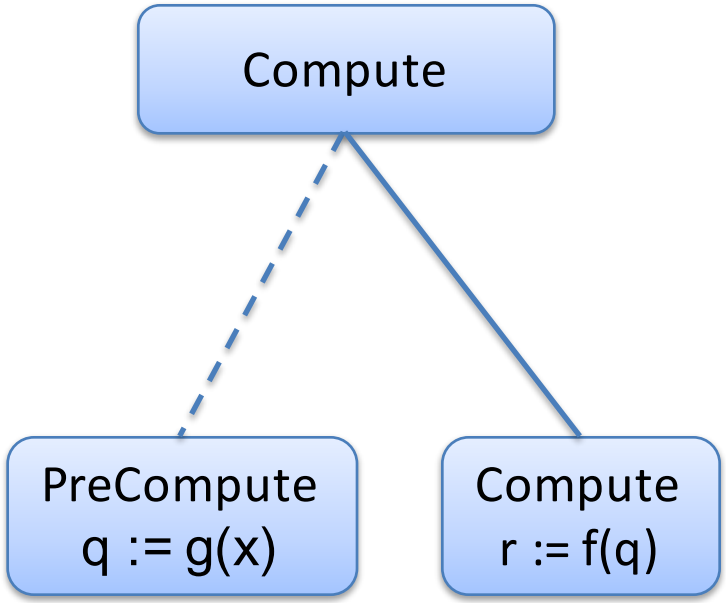
What's missing?

- Determining the refined events is relatively easy once we decide on the appropriate abstract program structure to use
- But determining the right gluing invariants can be difficult
- **Patterns**: from a certain specification structure
  - determine refined events *and*
  - gluing invariants

PreCompute Pattern

Compute ≙
begin
r := f(g(x))
end

Compute ≙
any q **where**
q = g(x)
then
r := f(q)
end



variable q

invariant
PreCompute ∧ ¬Compute
⇒ q = g(x)

The price: other events must maintain this invariant

General PreCompute Pattern

Compute ≙

any v1,v2 **where**

Goal1(v1,x)

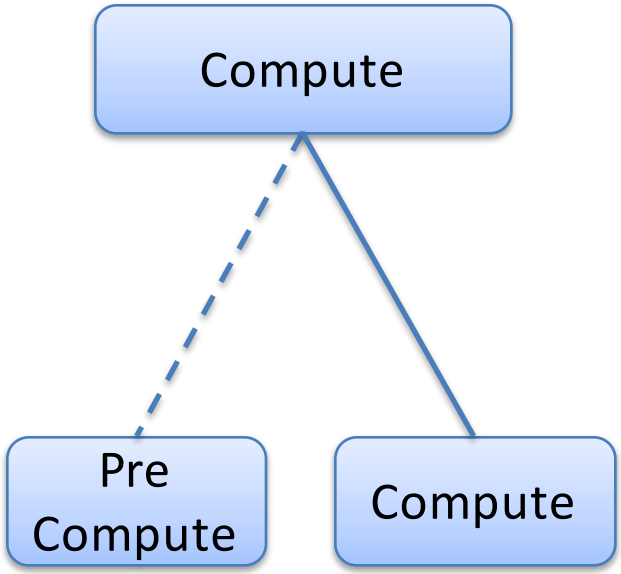
Goal2(v1,v2,x)

then

r := v2

end

Refinement of Compute



invariant

PreCompute ∧ ¬Compute
⇒ Goal1(v1,x)

Variable v1

PreCompute ≙
any v1' **where**
Goal1(v1',x)
then
v1 := v1'
end

Compute ≙
any v2 **where**
Goal2(v1,v2,x)
then
r:= v2
end

All Condition Pattern

CondAll ≙

when

$\forall i \in S \bullet Cond(i)$

then

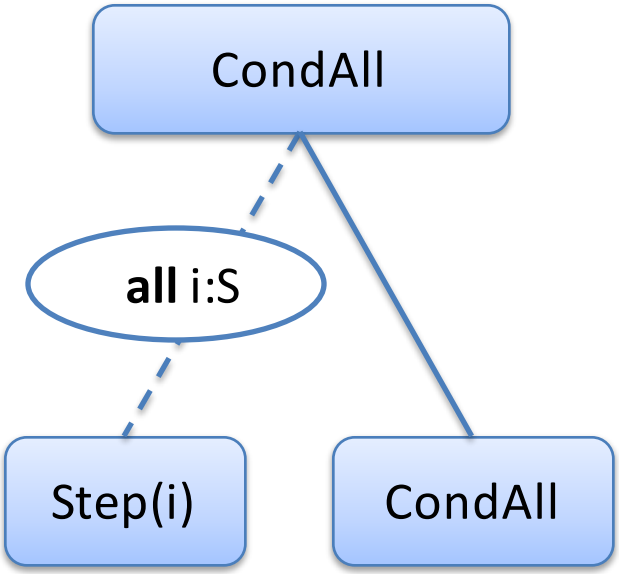
action

end

Example: search Fail:

Cond(i) ≙ ¬Property(i)

Refinement of CondAll



Step ≙
any i **where**
Cond(i)
end

CondAll≙
begin
action
end

invariant
$\forall i \in Step \bullet Cond(i)$

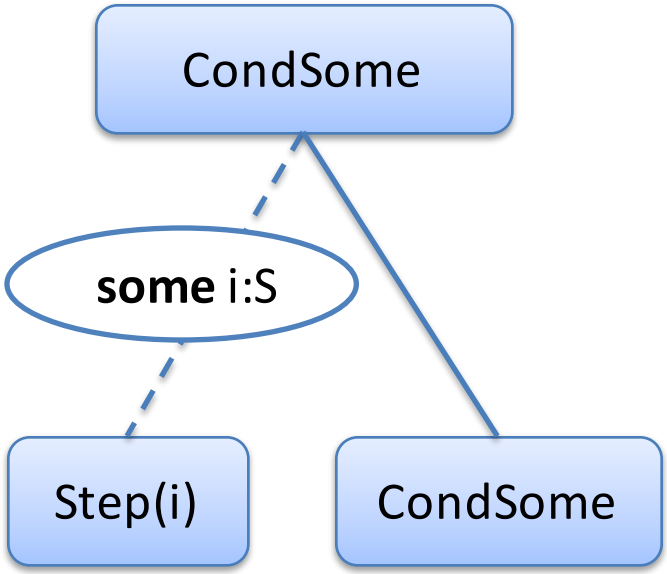
Some Condition Pattern

CondSome ≙
  when
    $\exists$ i . i $\in$ S $\wedge$ Cond(i)
  then
    action
  end

More usually written:

CondSome ≙
  any i **where**
    i $\in$ S
    Cond(i)
  then
    action(i)
  end

Refinement of CondSome

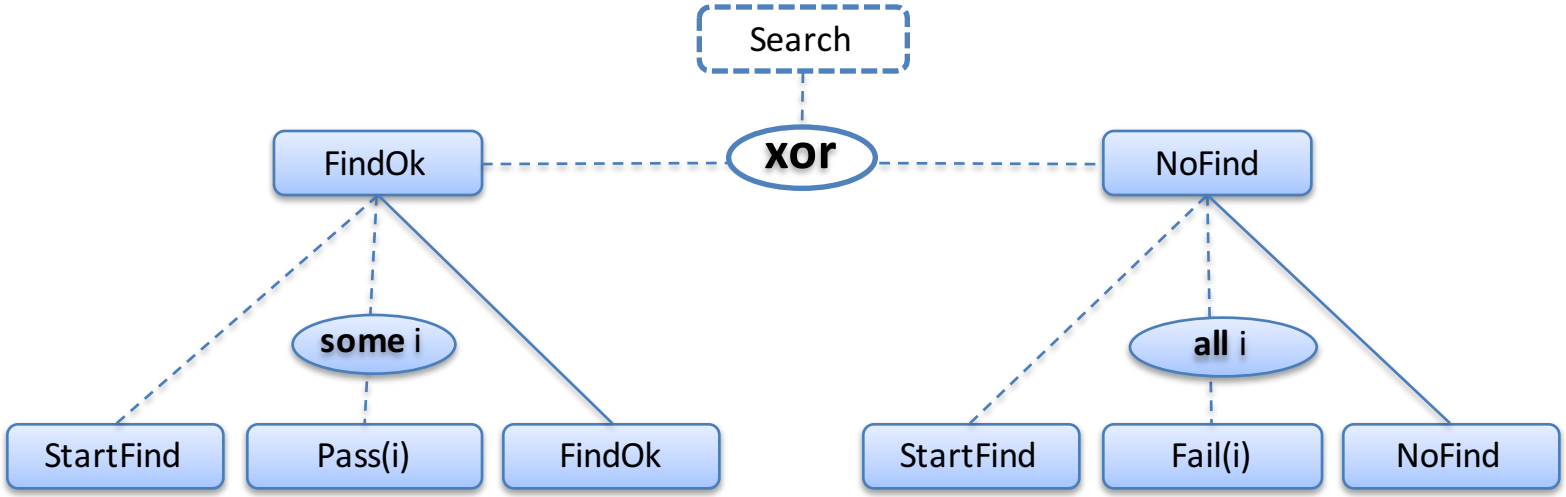


Step ≙
any i **where**
Cond(i)
end

CondSome ≙
any i **where**
i ∈ Step
then
action(i)
end

invariant
∀ i ∈ Step • Cond(i)

Refinement invariants for search



- $\forall i \in Pass$ • Property(i)
- $\forall i \in Fail$ • ¬Property(i)

PreCompute Map Pattern

sets

I // *indices*

T // *values*

variables

$M \in I \rightarrow T$

ComputeMap ≙

// compute map satisfying goal

any M' **where**

$\forall i \in I \bullet Goal(i, M'(i))$

then

M := M'

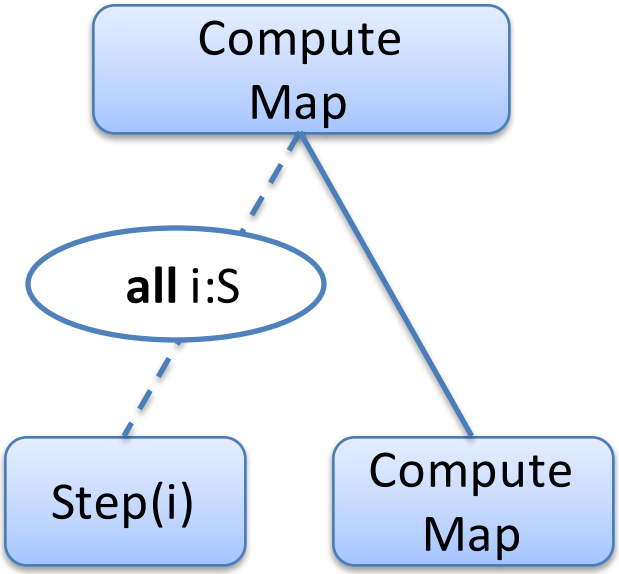
end

File write:

disk is the map to be computed

Goal(i, dsk(i)) ≙ dsk(i) = buffer(i)

Refinement of PreComputeMap



Step ≙
  any i, v **where**
    Goal(i, v)
  then
    M(i) := v
  end

ComputeMap ≙
  begin
    skip
  end

invariant
$\forall i \in Step \bullet Goal(i, M(i))$

UpdateMap

sets

I

T

variables

$M \in I \rightarrow T$

UpdateMap ≙

any M' **where**

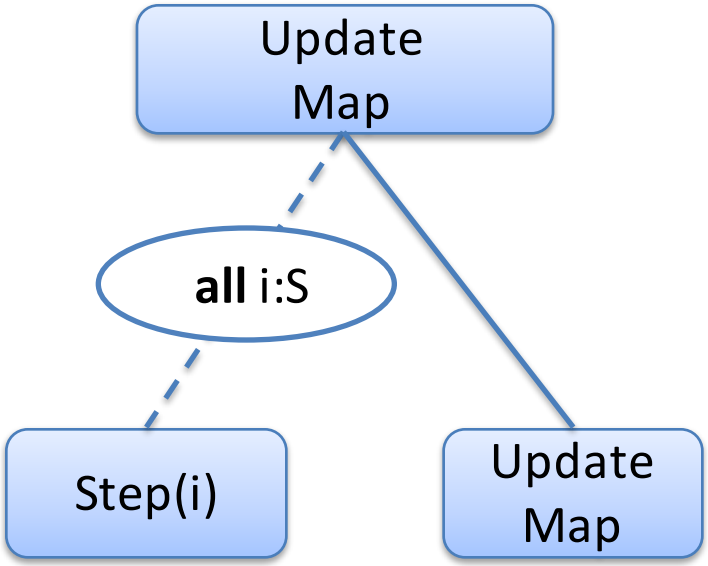
$\forall i \in I \bullet M'(i) = f(M(i))$

then

M := M'

end

UpdateMap Pattern



Step ≙
  any i **then**
    N(i) := f(N(i))
  end

UpdateMap ≙
  begin
    skip
  end

invariant

$$\forall i \in S \bullet$$
$$i \notin Step \lor UpdateMap \Rightarrow N(i) = M(i)$$
$$i \in Step \land \neg UpdateMap \Rightarrow N(i) = f(M(i))$$

MapReduce

event **OutputReductions**

any j t1 t2 t3 mws rws

where

t1 = **split**(j)

$\forall$ m · t2(m) = **map**(t1(m))

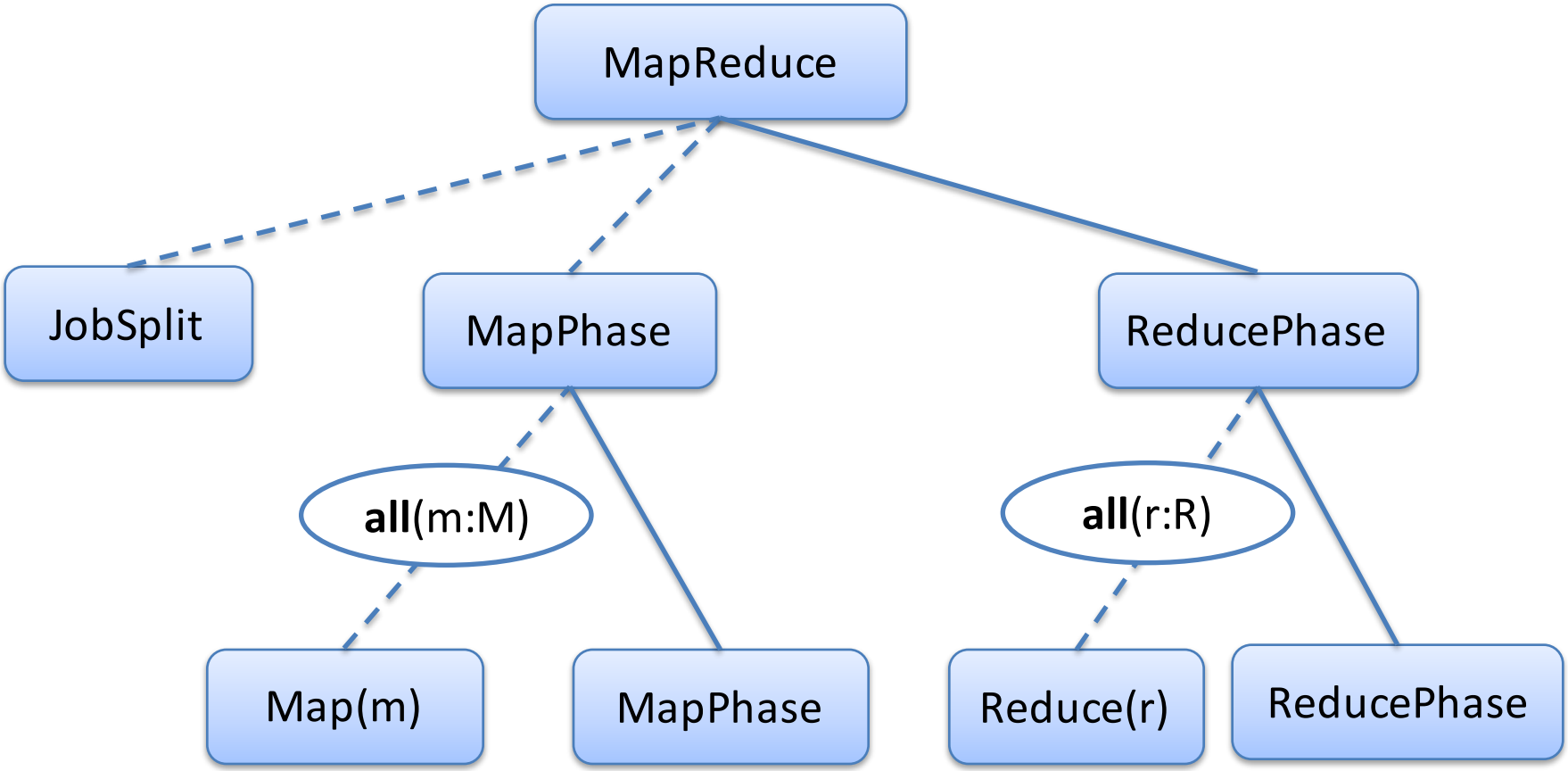
$\forall$ r· t3(r) = **reduce**({ m· t2(m)(r) })

then

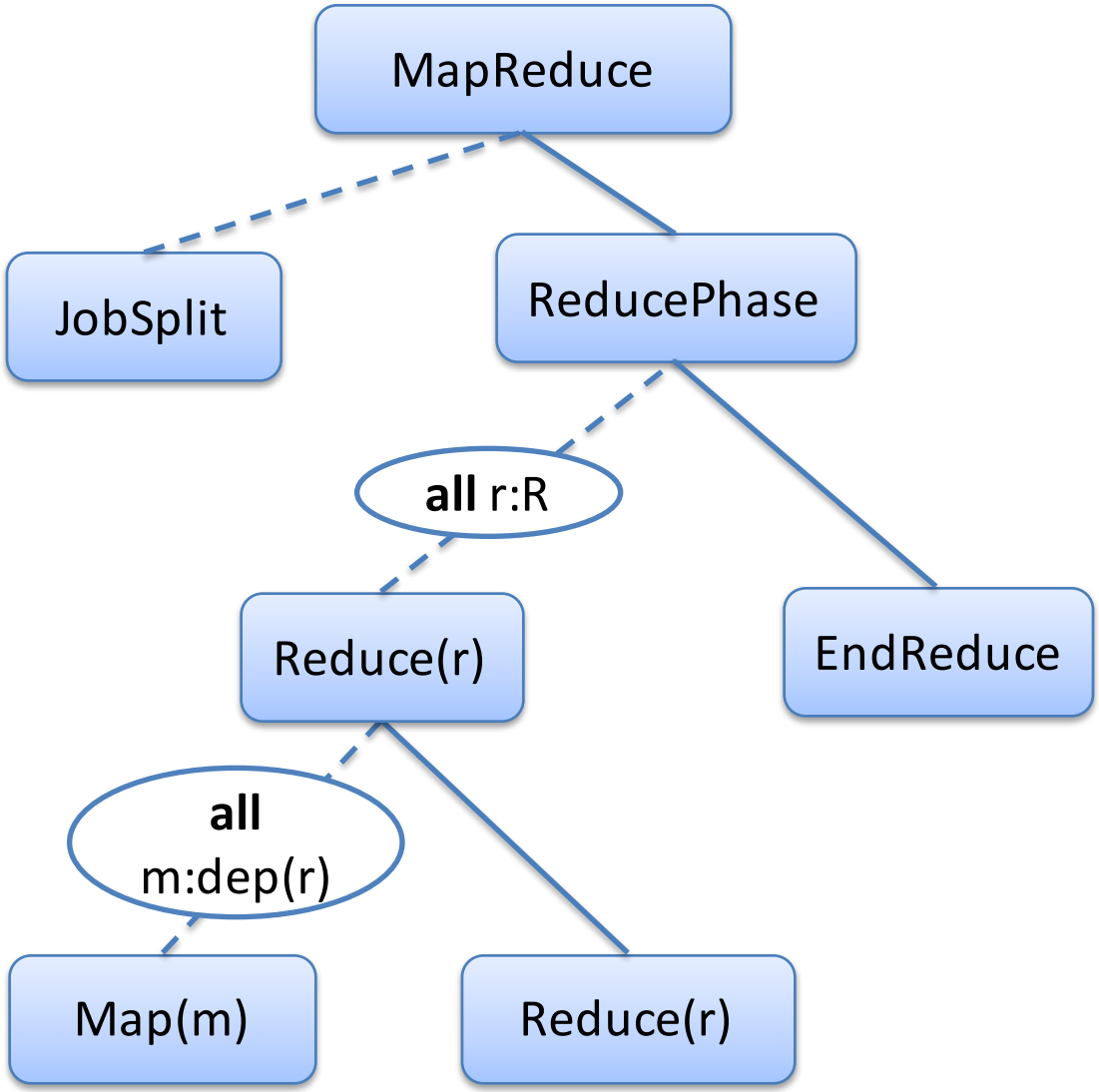
output ≔ **combine**({r · t3(r)})

end

MapReduce


MapReduce
JobSplit
MapPhase
ReducePhase
all(m:M)
all(r:R)
Map(m)
MapPhase
Reduce(r)
ReducePhase

MapReduce (less synchronised)



Some References

- Butler, *Decomposition Structures for Event-B*. In, Integrated Formal Methods iFM2009.

- Salehi, Butler, Rezazadeh, *Language and tool support for event refinement structures in Event-B*. Formal Aspects of Computing, 27, (3), 2015.

- Dghaym, Butler, Salehi, *Evaluation of graphical control flow management approaches for Event-B modelling*. Intl Workshop on Automated Verification of Critical Systems (AVocS 2013).

- Pereverzeva, Butler, Salehi, Laibinis, Troubitsyna, *Formal derivation of distributed MapReduce*. ABZ 2014 Conference, Toulouse, June 2014.

Rodin Toolset for Event-B
www.event-b.org

- Open source Eclipse based
- Proof obligation generation
- Customisable use of internal and external provers and model checkers
- Extensible theories
- Value-added modelling front ends
  - UML-B, Model Visualisation, Co-simulations,…

Concluding

- Abstract program structures add value to existing refinement framework
  - Structures provide explicit representation of atomicity decomposition (with sufficient interleaving)
  - Power of diagrams – rapid understanding
- Not quite transformational approach:
  - abstract programs provide structures for refining models
  - problem-based patterns for introducing abstract programs in a verifiable way

End