

Relating problem structure and solution structure in Event-B refinement

Michael Butler

WG 2.3 Meeting, Winchester



Problem versus Solution

- **Problem** decomposition:
 - In order to understand a complex problem, break it into sub-problems and master each sub-problem
 - Sub-problem should have a degree of 'unity'
(Michael Jackson)
- **Solution** decomposition:
 - Structure an implementation of a system as an assembly of interacting components
 - Follow architectural principles
- Problem structure may be **orthogonal** to solution structure

Example: Microproc ISA

(John Colley, PhD thesis 2010)

- Problem decomposition

- Arithmetic instructions



- Memory access instructions



- Branching instructions



- Solution decomposition (pipeline arch.)

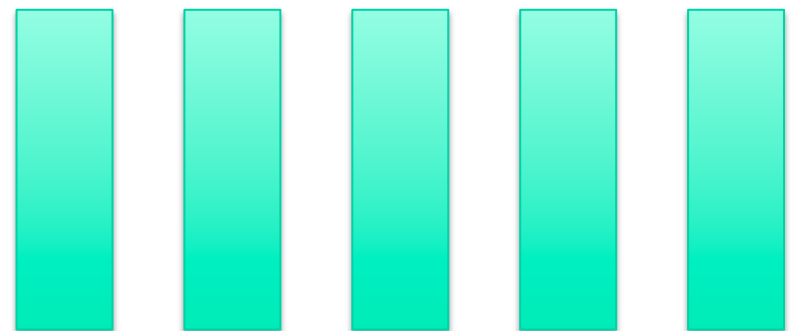
- Stage 1: Inst fetch

- Stage 2: Inst decode

- Stage 3: Inst execute

- Stage 4: Memory access

- Stage 5: Writeback (to registers)



- These structures are orthogonal

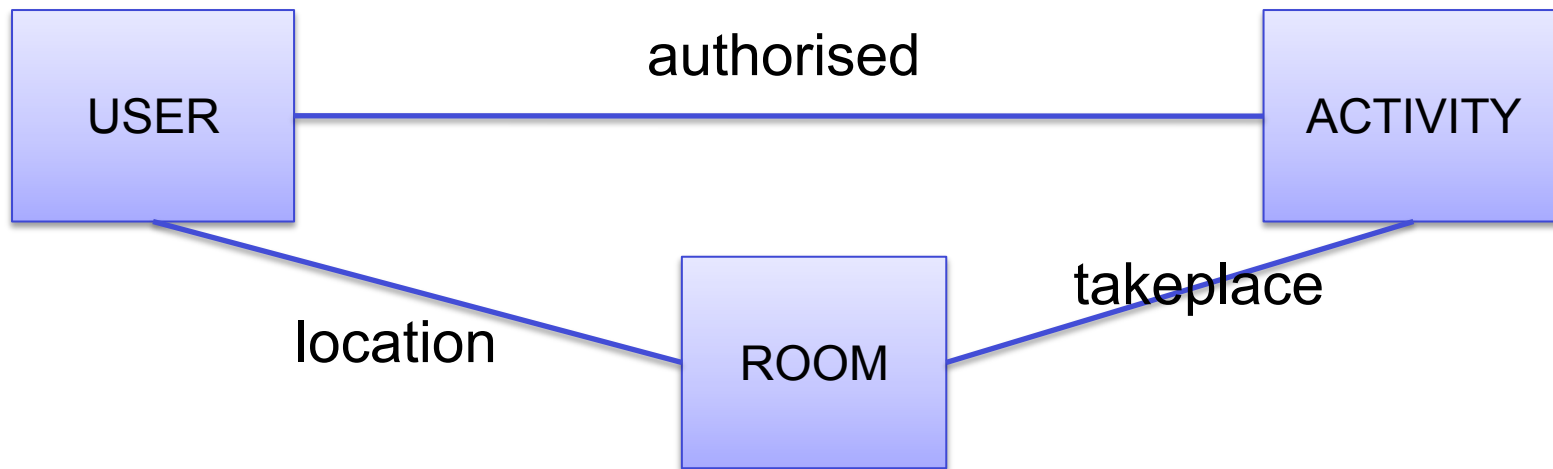
Outline

- Event-B
 - Modelling
 - Refinement
 - Rodin tool
 - Decomposition
- Problem decomposition and reconciliation
- Transforming to solution structure

Event-B (Abrial)

- **State-based** (like ASM, B, VDM, Z)
 - **events**: guarded atomic assignments
 - **set theory** as mathematical language
- **Refinement**:
 - refinement at level of events
 - gluing invariants
 - based on Action Systems (Back)
- **Proof method** supported by **Rodin tool**
 - Refinement proof obligations (POs) generated from models
 - Automated and interactive theorem provers for POs

Access control example



Variables of Event-B model

@inv1 authorised \in User \leftrightarrow Activity // relation

@inv2 takeplace \in Room \leftrightarrow Activity // relation

@inv3 location \in User \rightarrow Room // partial function

Variables and invariants of Event-B model

Variables of Event-B model

@inv1 authorised \in User \leftrightarrow Activity // relation
@inv2 takeplace \in Room \leftrightarrow Activity // relation
@inv3 location \in User \rightarrow Room // partial function

Access control *invariant*:

if user u is in room r ,

then u must be authorised to engaged in all activities that can take place in r

@inv4 $\forall u, r . u \in \text{dom}(\text{location}) \wedge \text{location}(u) = r \Rightarrow$
 $\text{takeplace}[r] \subseteq \text{authorised}[u]$

Rodin demo

Rodin Open Tool Platform

Extension of Eclipse IDE

Repository of structured modelling elements

Rodin Eclipse Builder manages:

Well-formedness + type checker

Consistency/refinement Proof Obligation generator

Proof manager

Propagation of changes

Extension points

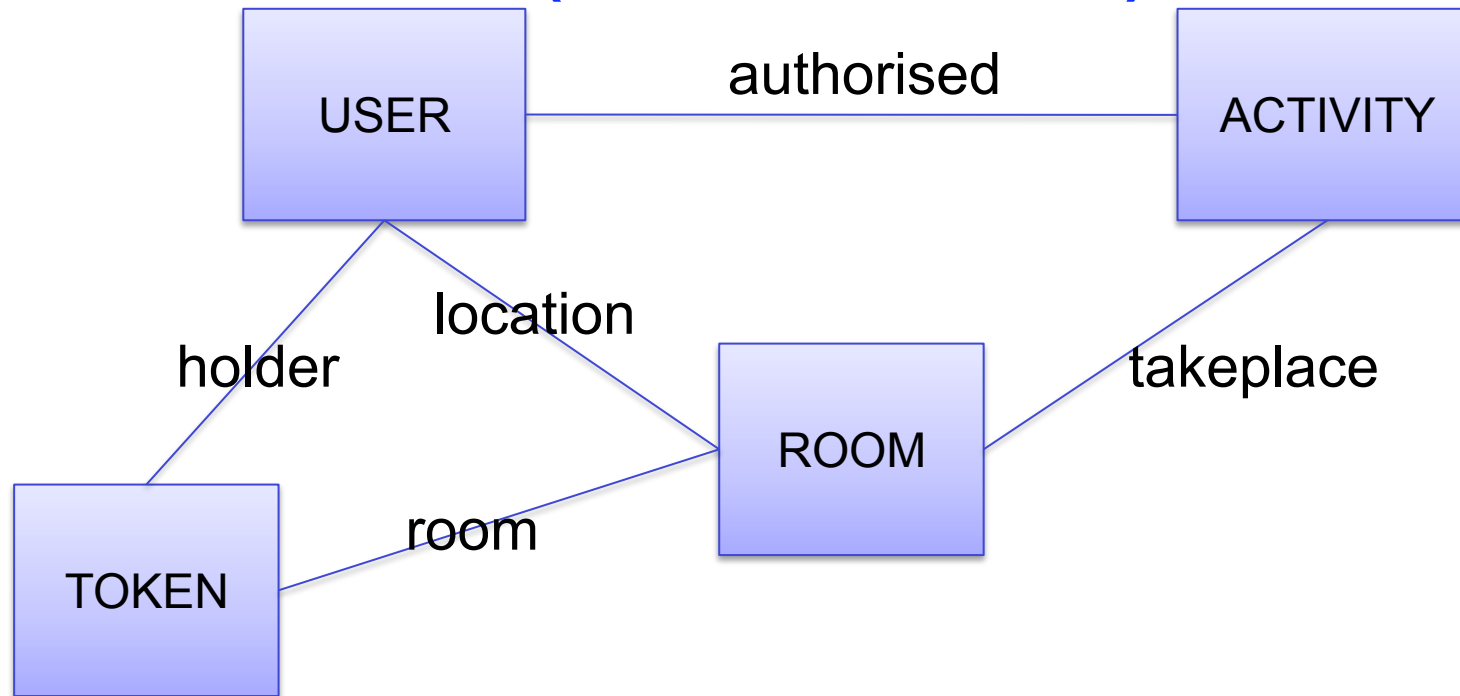
Open source

www.event-b.org

Refinement

- Refinement: process of **enriching** or **transforming** a model in order to
 1. **augment the functionality** being modelled, **or**
 2. **explain how** some purpose is achieved
- Consistency of a refinement:
 - We use **proof** to verify the **consistency** of a refinement step
 - **Failing proof** can help us identify **inconsistencies** in a refinement step

We construct a new model (refinement)



Abstract guard on a user and room for entering

grd3: $\text{takeplace}[r] \subseteq \text{authorised}[u]$

is replaced by a guard on a token

grd3b: $t \in \text{valid} \wedge \text{room}(t) = r \wedge \text{holder}(t) = u$

Feature augmentation: layered specification of Flash-based filestore

ML0 : Tree properties and basic operations affecting tree structure

ML1: Partition *objects* into *files* and *directories*

ML2 : Introduces file content

ML3: Introduces permissions

ML4: Introduces other missing properties such as *name*, *date* of creation and last modification

Damchoom, K., Butler, M. and Abrial, J. R.

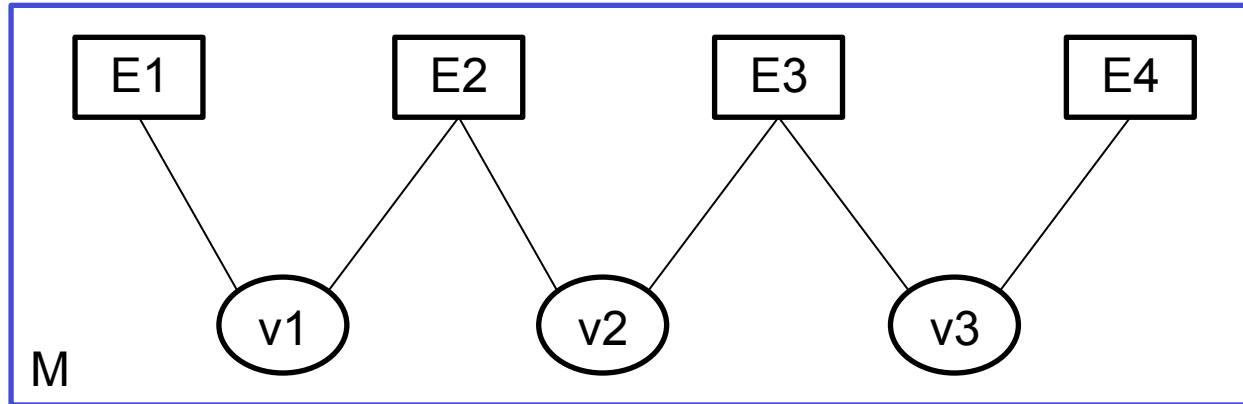
Modelling and proof of a Tree-structured File System in Event-B and Rodin.

ICFEM 2008

Decomposition in Event-B

- **Shared variable** decomposition
 - strong coupling between machines (**weak encapsulation**)
 - data refinement of shared variables is difficult
- **Shared event** decomposition
 - embodies **strong encapsulation**
 - data refinement of encapsulated variables is easy

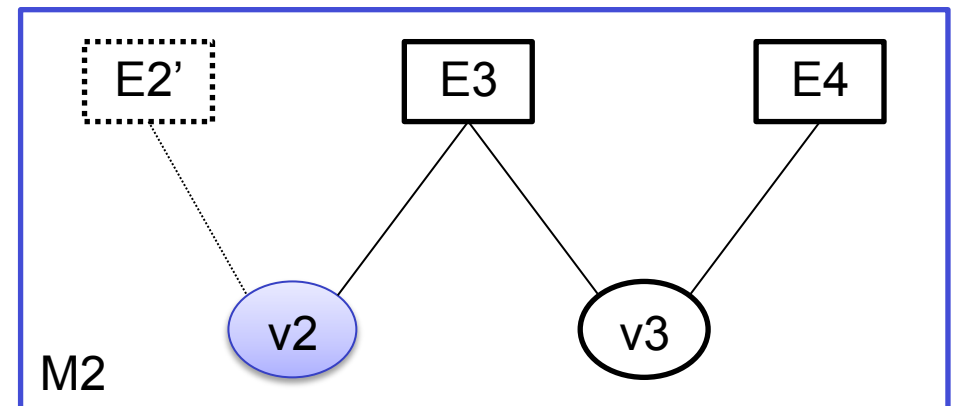
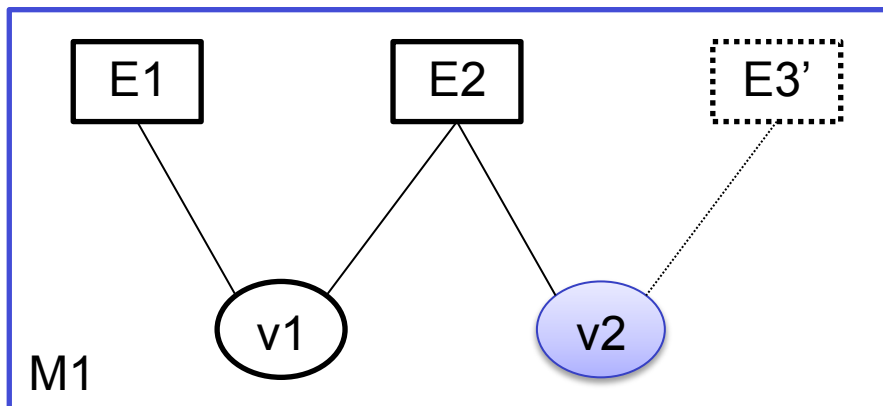
Shared Variable Decomposition



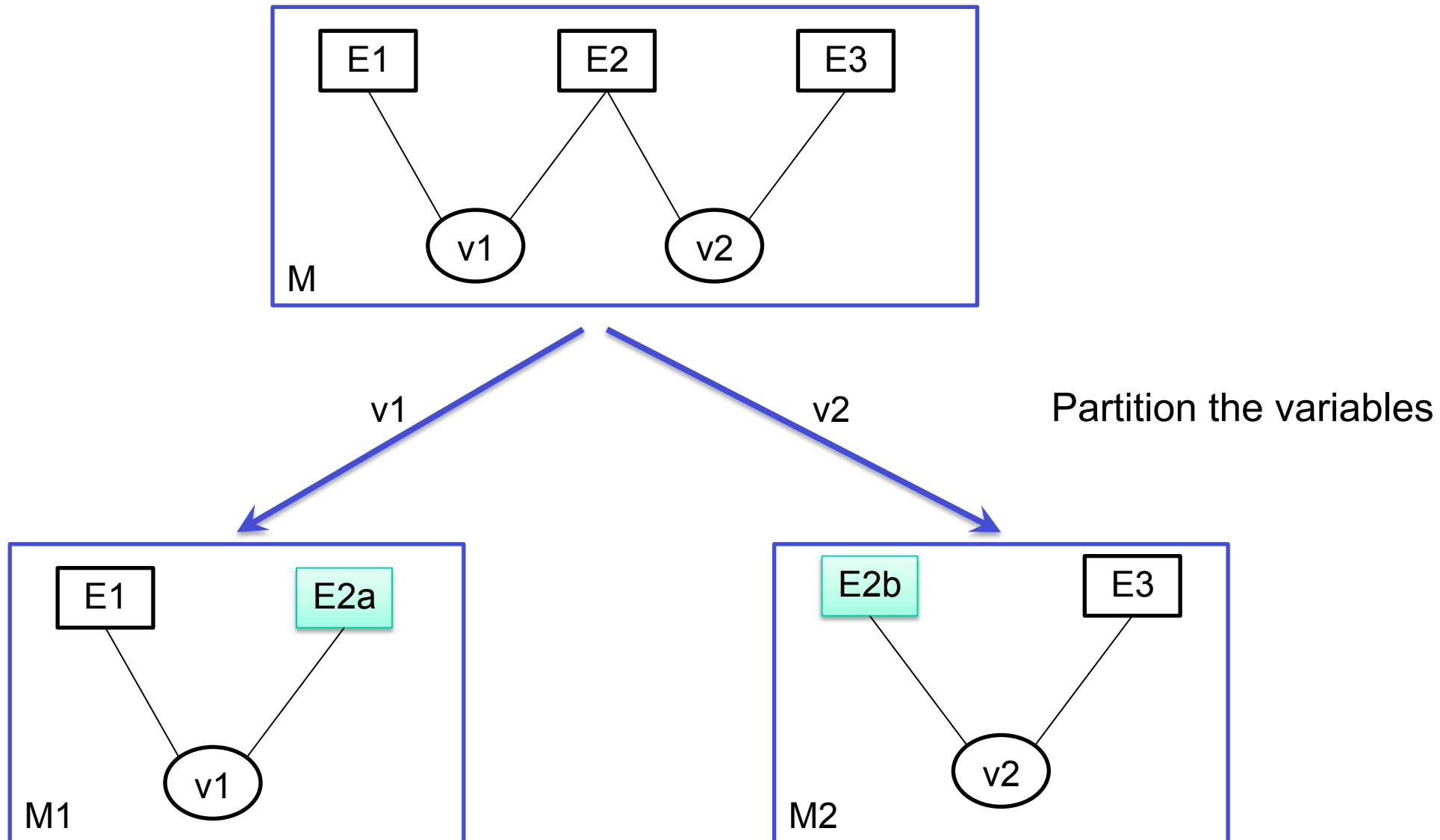
E1, E2

E3, E4

Partition the events



Shared Event Decomposition



Refinement after decomposition

- Shared event: can refine sub-model provided
 - Common parameters of shared events are maintained
- Shared variable: can refine sub-model provided
 - External events are not refined (rely condition)
 - Shared variables are not refined.
 - Invariants used in refinement are preserved by external events

Partitioning events for sharing

```
E =  
  any p where  
    G1( x, p )  
    G2( y, p )  
  then  
    x := H1( x, p )  
    y := H2( y, p )  
  end
```

```
Ex =  
  any p where  
    G1( x, p )  
  then  
    x := H1( x, p )  
  end
```

```
Ey =  
  any p where  
    G2( y, p )  
  then  
    y := H2( y, p )  
  end
```

Simple value transfer example

variable x, y

invariant $x + y = N$

init $x := 0 \parallel y := N$

event TRANSFER

any a **where**

$a \in \mathbb{Z}$

$y \geq a$

then

$x := x + a$

$y := y - a$

end

Decompose

X1	Y1
<p>variable x</p> <p>invariant $x \in \mathbb{Z}$</p> <p>init $x := 0$</p> <p>Event INC any a where $a \in \mathbb{Z}$ then $x := x + a$ end</p>	<p>variable y</p> <p>invariant $y \in \mathbb{Z}$</p> <p>init $y := N$</p> <p>Event DEC any a where $a \in \mathbb{Z}$ $y \geq a$ then $y := y - a$ end</p>

Pre-partitioning

E =

any p **where**

G1(x, p, f(y))

G2(y, p)

then

x := H1(x, p, f(y))

y := H2(y, p)

end

E =

any p, q **where**

q = f(y)

G1(x, p, q)

G2(y, p)

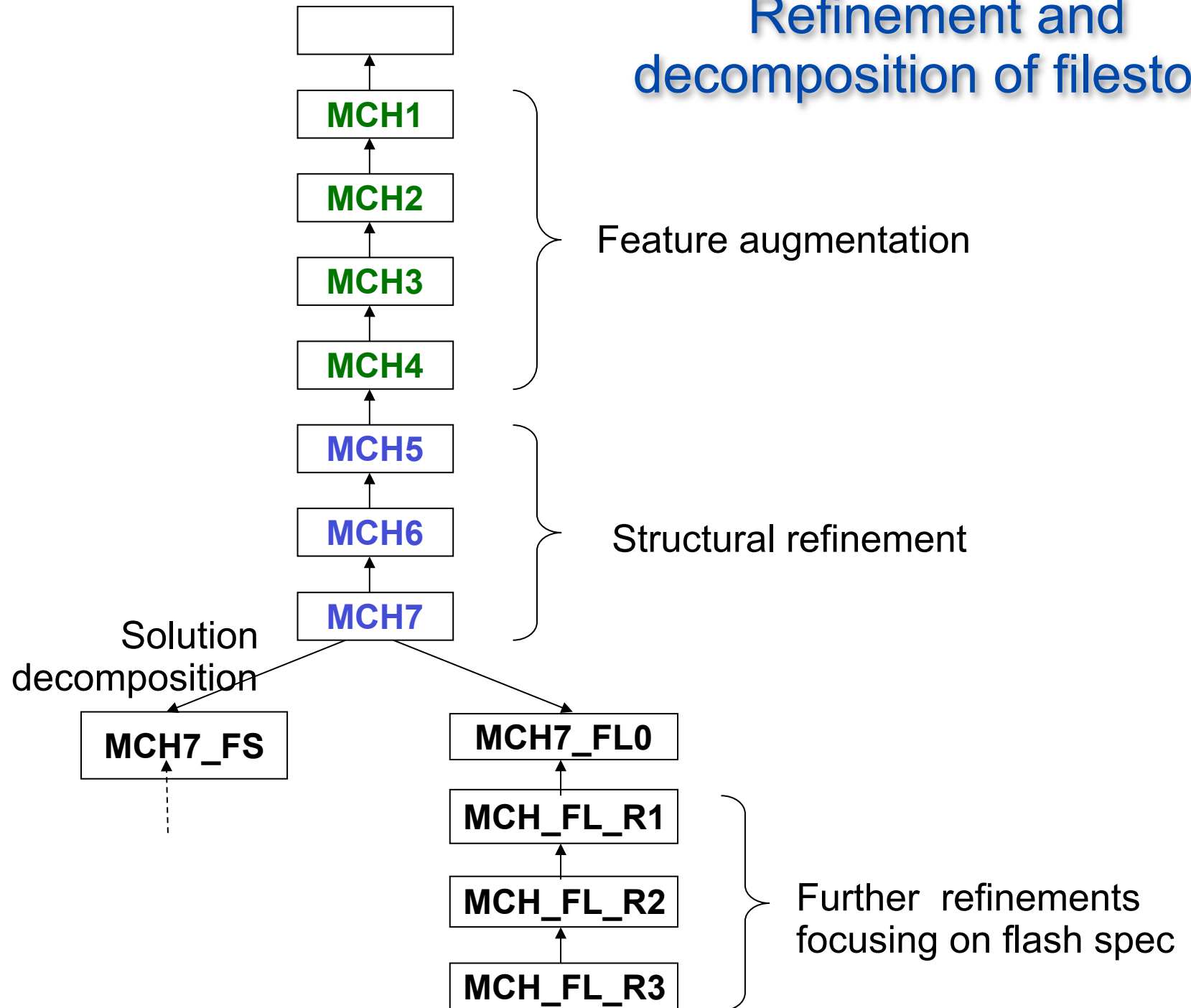
then

x := H1(x, p, q)

y := H2(y, p)

end

Refinement and decomposition of filestore



Structural refinement of FFS

ML5: Decomposes event *write* into
w_start, w_step, w_end (*ok, fail*)

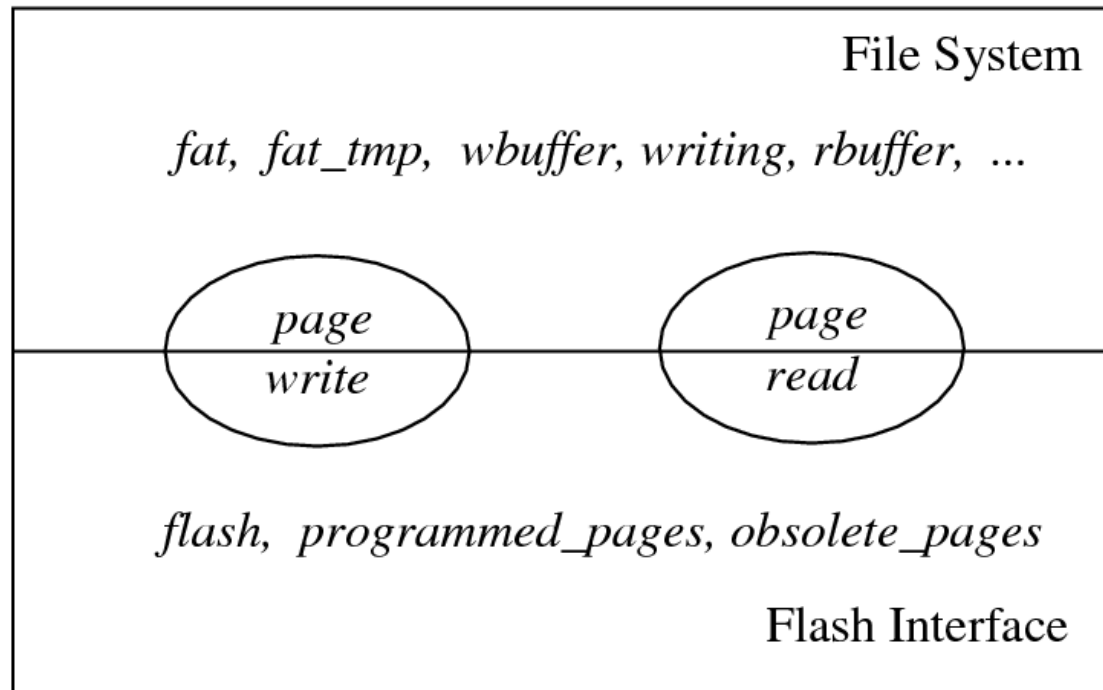
ML6 : Decomposes event *read* into
r_start, r_step, r_end (*ok, fail*)

ML7: Links the FS to the flash specification by
introducing flash properties

Solution Decomposition

Partitions the machine level 7 into two machines representing the file system layer (FS) and the flash interface layer (FL).

Diagram of the machine decomposition

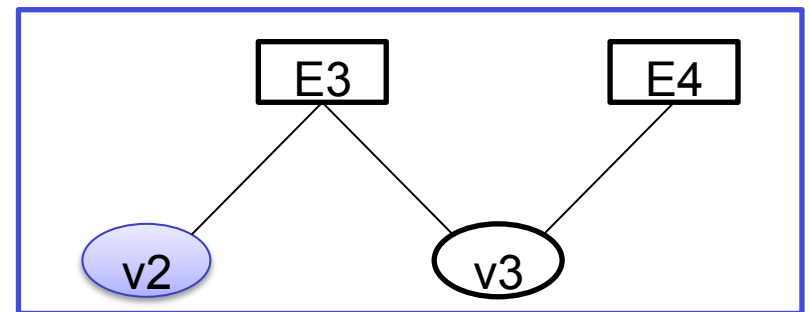
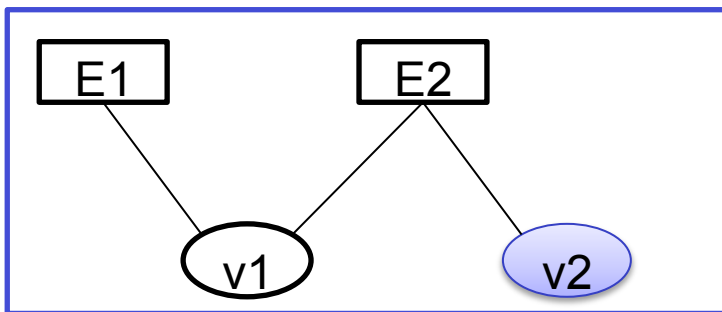


Supporting sub-problem structuring

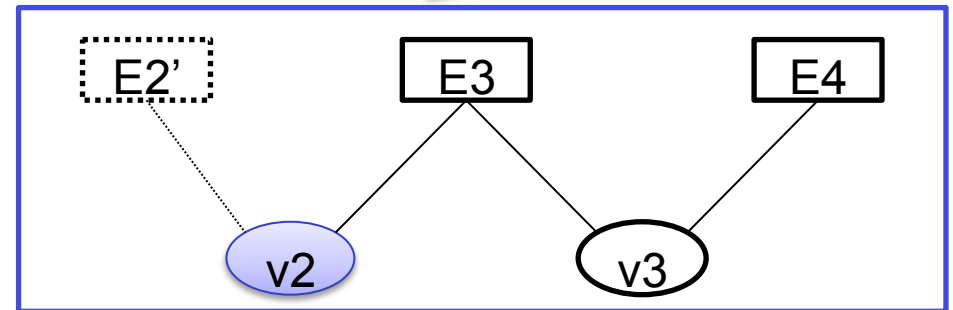
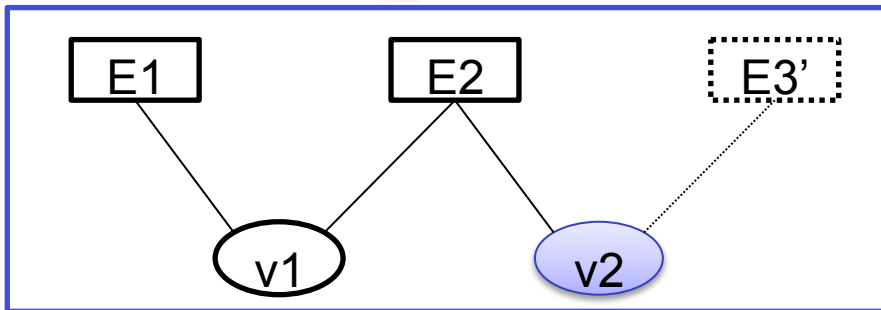
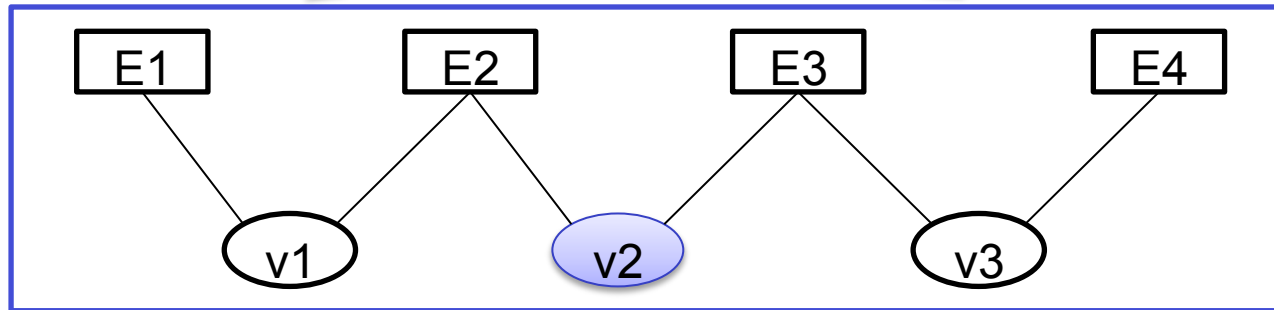
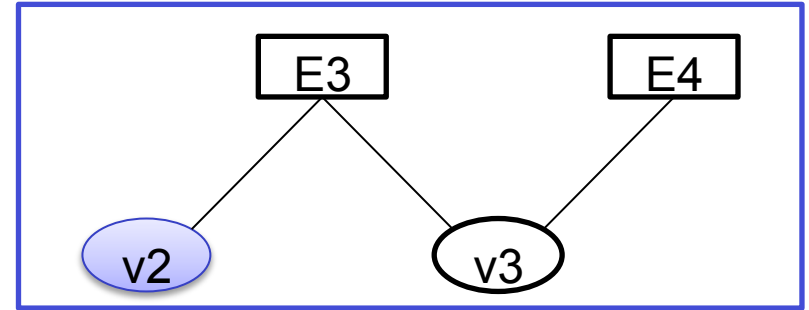
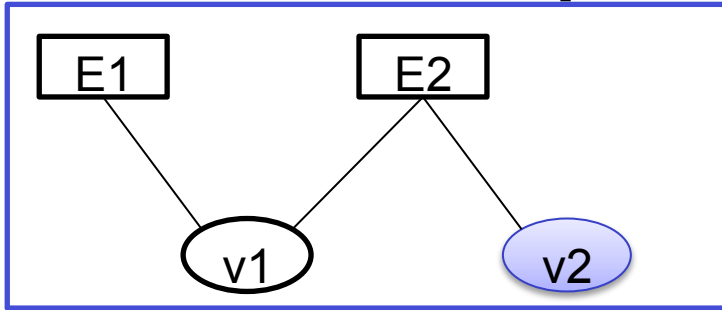
- Hypothesis:
 - Use **shared variable** for **problem** composition
 - Use **shared event** for **solution** decomposition

Sub-problems as Event-B machines

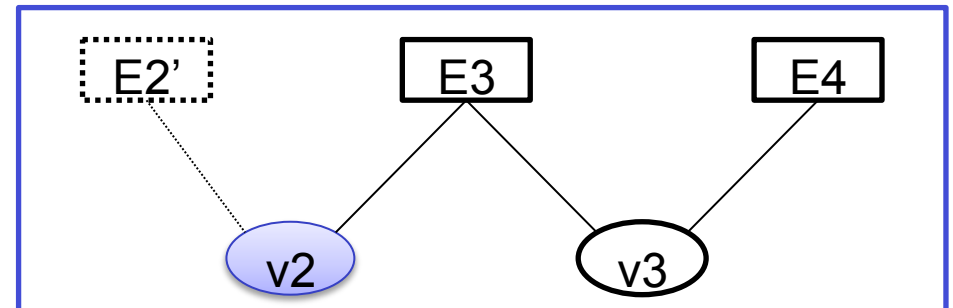
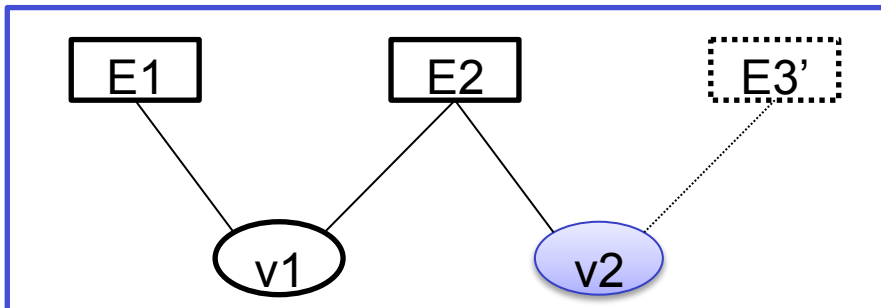
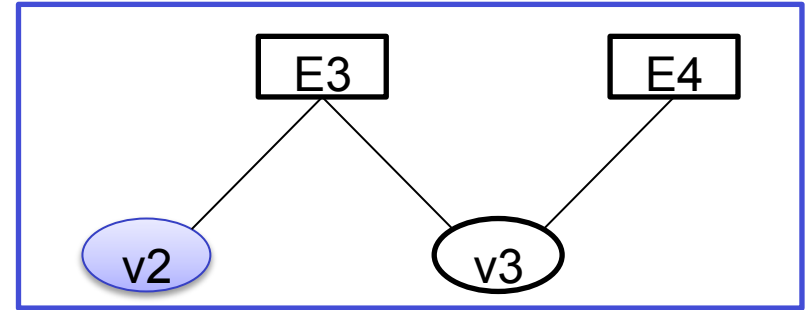
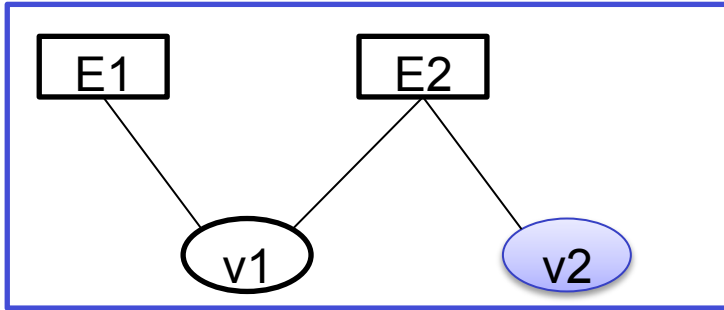
- Problem decomposition takes place prior to formalisation
 - sub-problems are identified informally
- Each sub-problem is then formalised
- Can the sub-problems be refined separately?



Sub-problems & separate refinement



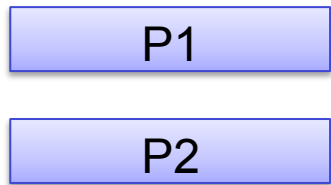
Sub-problem 'reconciliation'



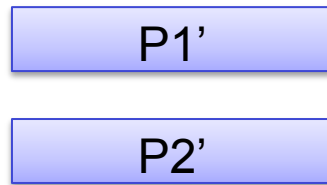
Problem decomposition and reconciliation

- In order to be able to refine sub-problems we ‘reconcile’ the sub-problem models
 - through addition of appropriate external events
 - reconciliation is formal
- Problem decomposition is informal
- Reconciliation of sub-models is formal

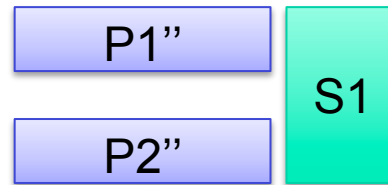
From problems to solution (staged)



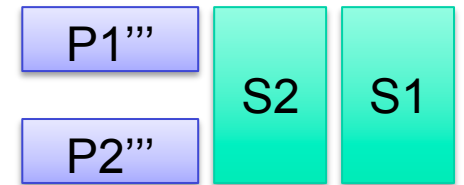
$P1 + P2$



$P1' + P2'$



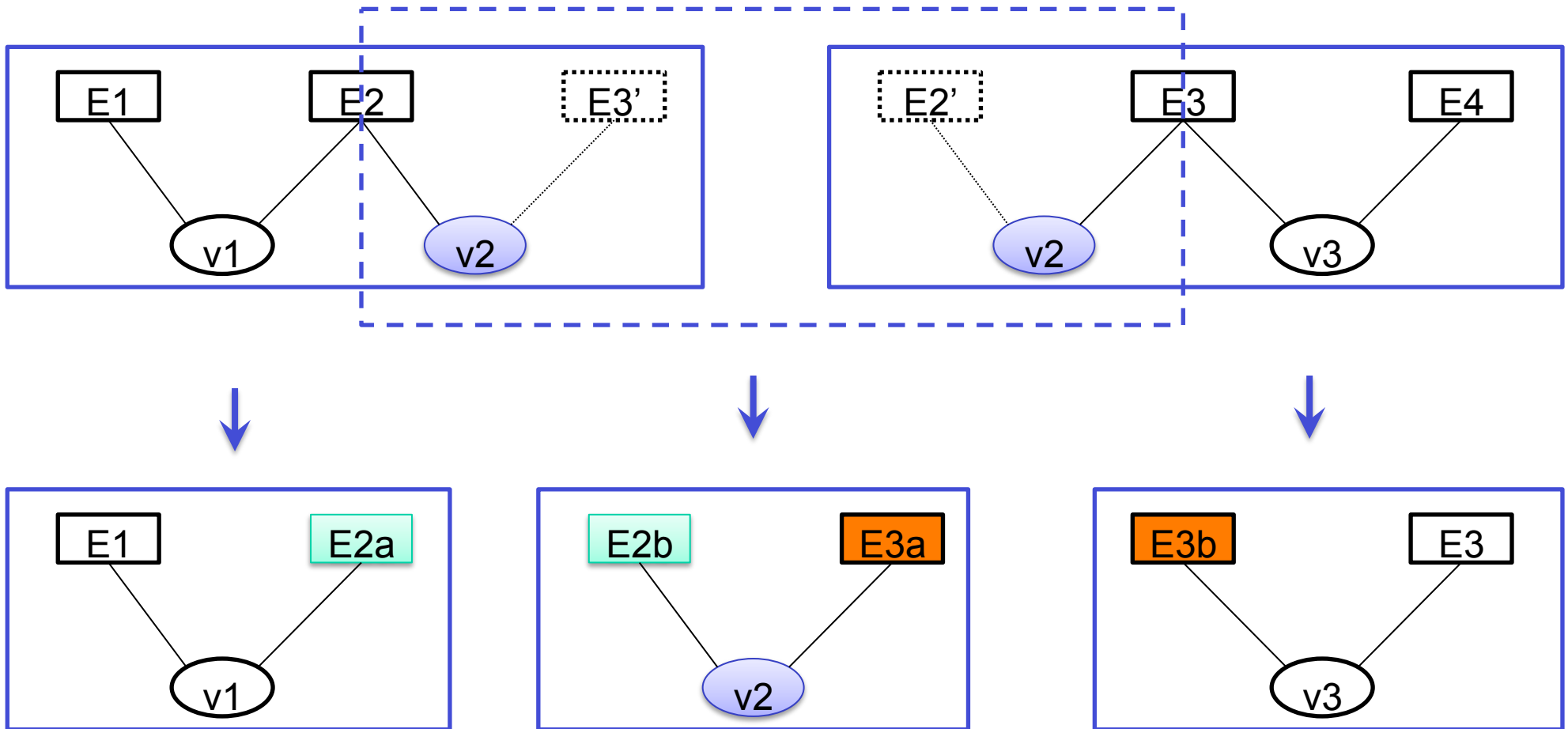
$(P1'' + P2'') \cdot S1$



$(P1''' + P2''') \cdot S2 \cdot S1$

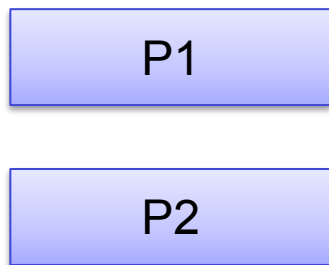
Factorisation

Factorisation



- Encapsulation means v2 can be data refined independently
- Factorisation should be delayed until refinement of subproblems no longer relies on v2

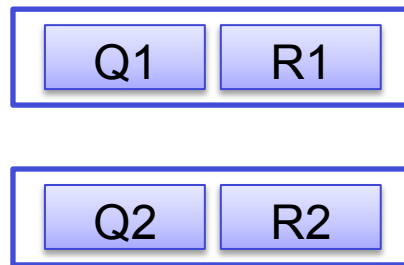
Exchange



$P1 + P2$

Q1 and Q2 share vars

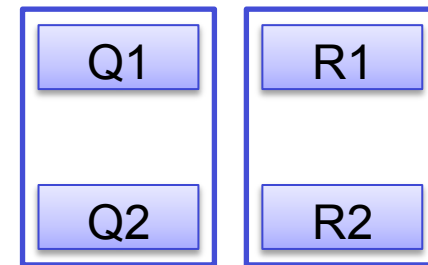
R1 and R2 share vars



$(Q1 \cdot R1) + (Q2 \cdot R2)$

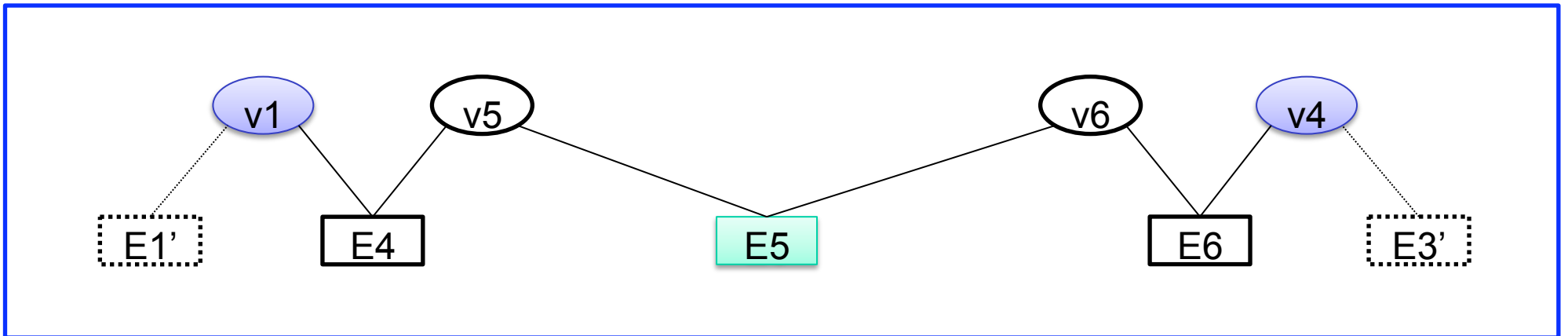
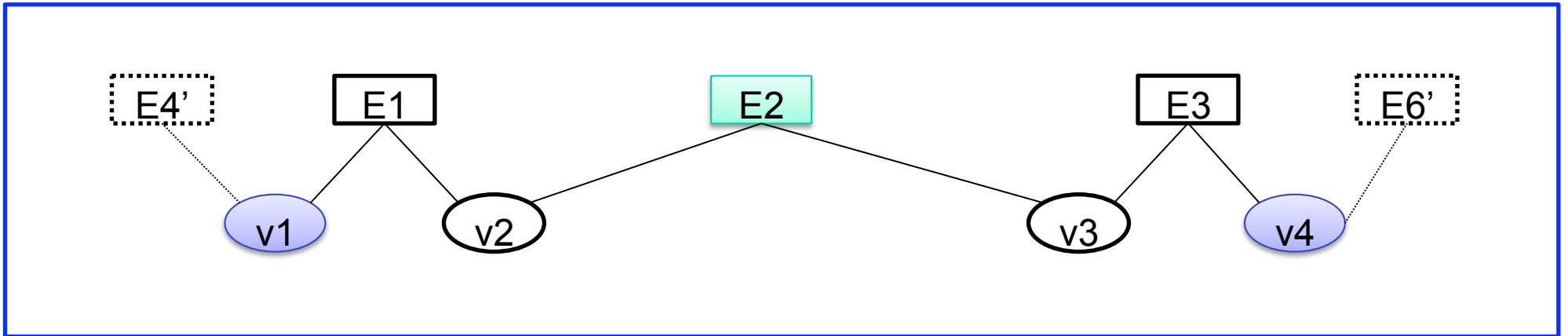
Q1 and R2 do not share vars

Q2 and R1 do not share vars

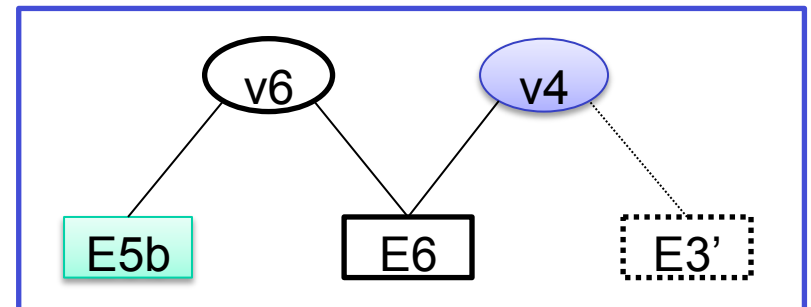
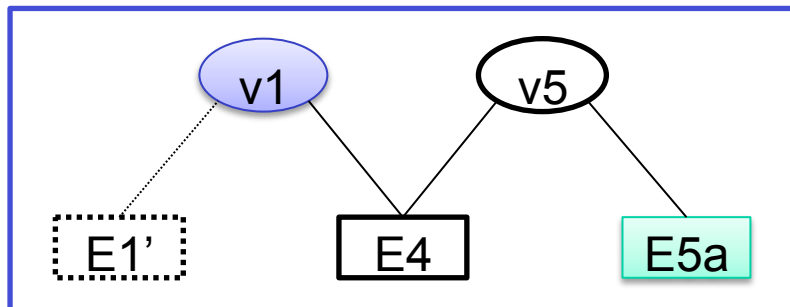
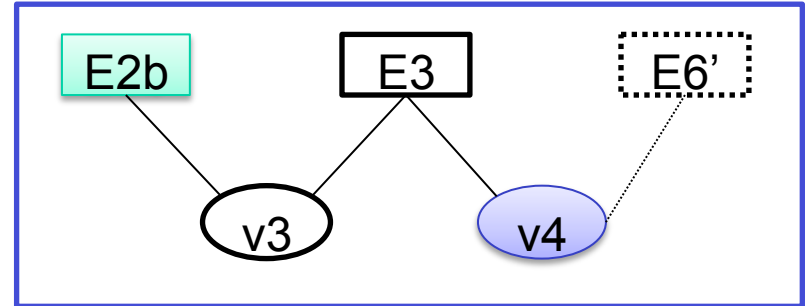
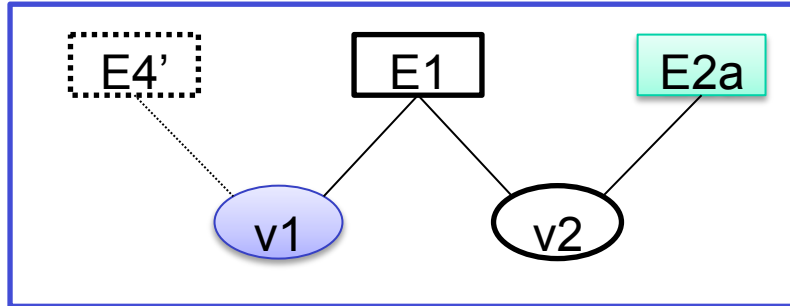


$(Q1 + Q2) \cdot (R1 + R2)$

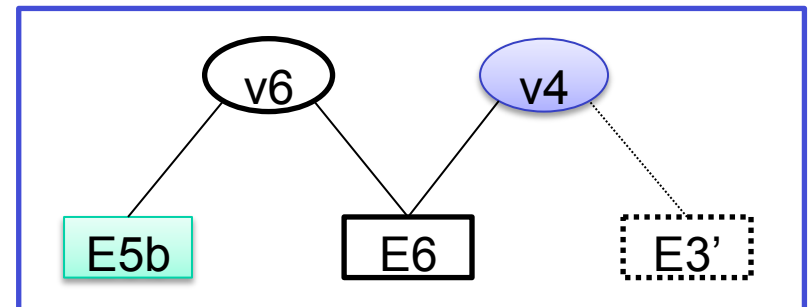
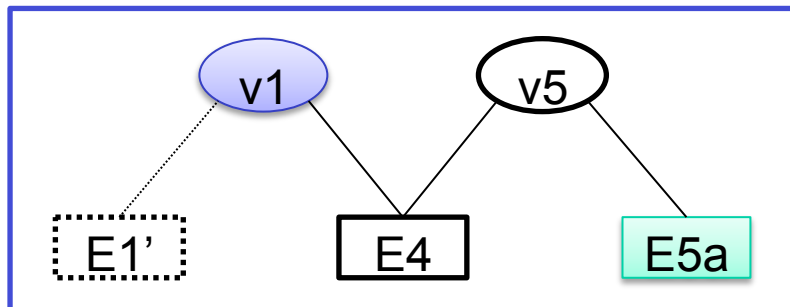
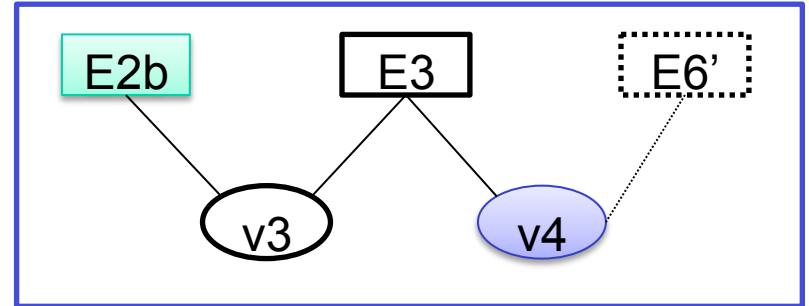
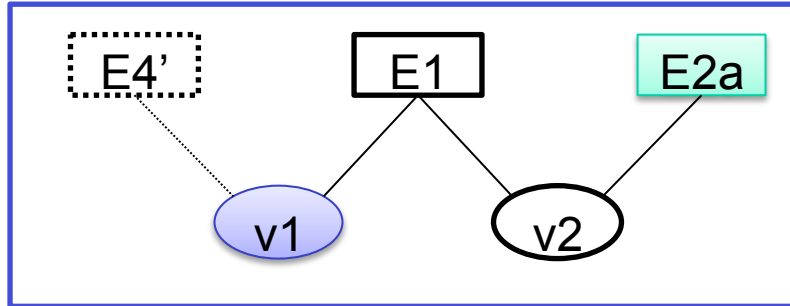
P1 + P2



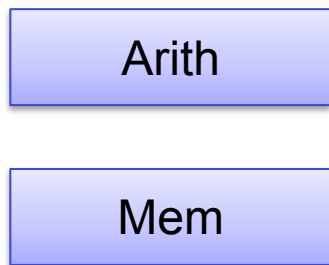
$$(Q1 \cdot R1) + (Q2 \cdot R2)$$



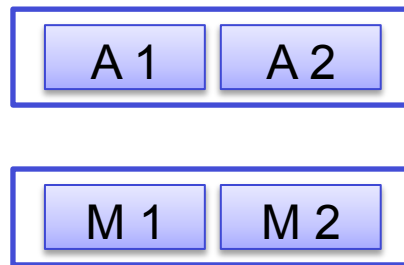
$$(Q1 + Q2) \cdot (R1 + R2)$$



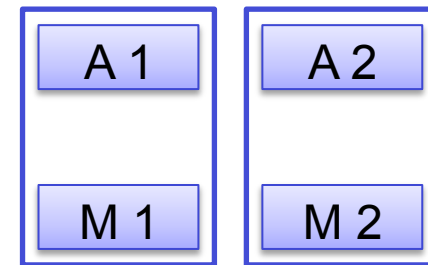
Exchange in pipeline architecture



$$A + M$$

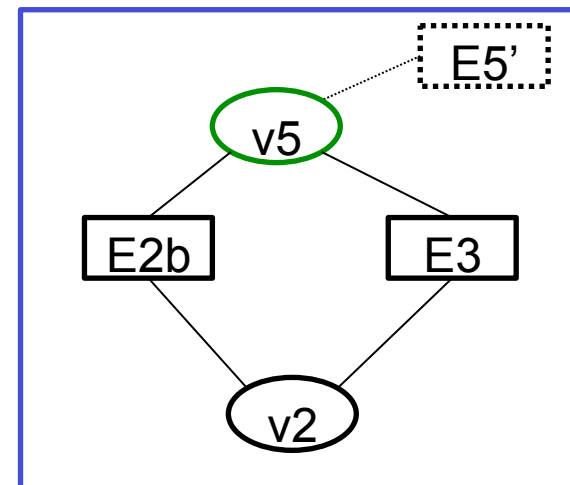
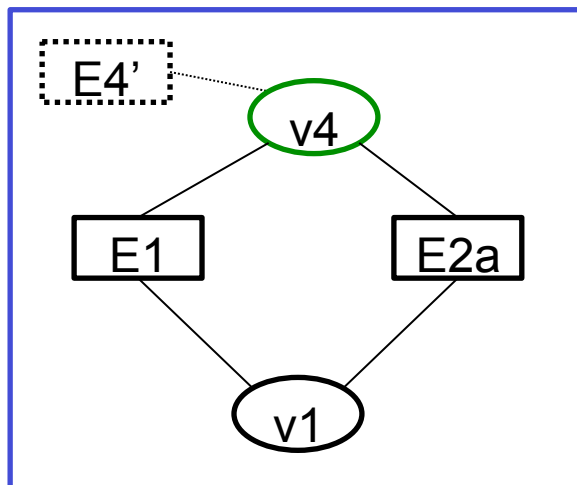
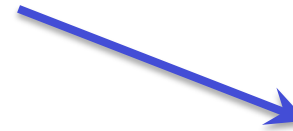
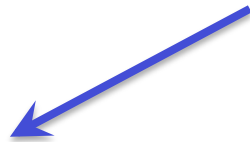
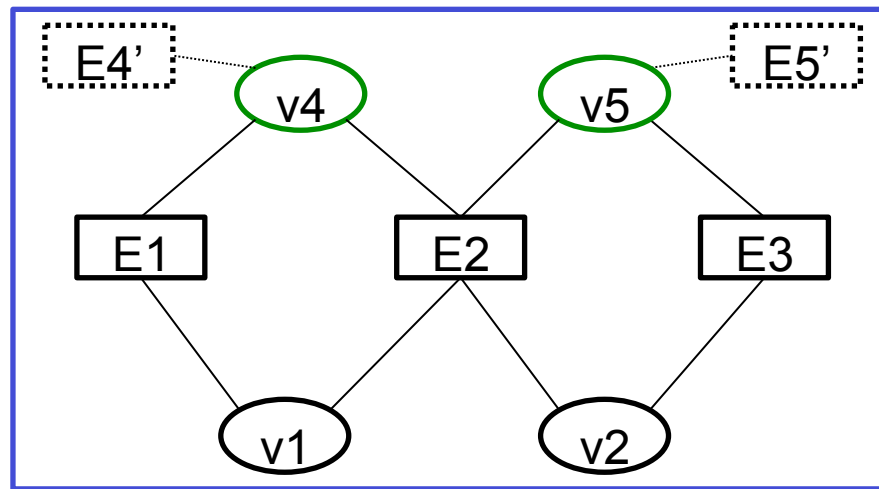


$$(A1 \cdot R1) + (A2 \cdot M2)$$



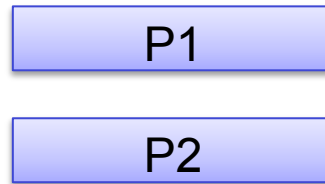
$$(A1 + M2) \cdot (A2 + M2)$$

Shared event decomposition (with external variables and events)

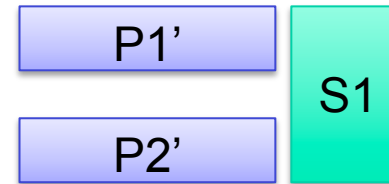


Structure refactoring

Factorisation:



$$P1 + P2$$

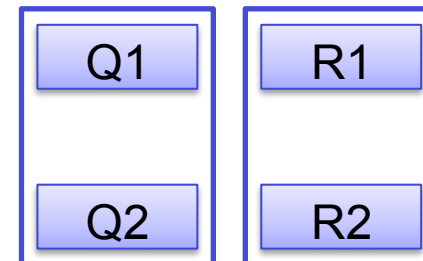


$$(P1' + P2') \cdot S1$$

Exchange:



$$(Q1 \cdot R1) + (Q2 \cdot R2)$$



$$(Q1 + Q2) \cdot (R1 + R2)$$

What's needed in tooling?

- Explicit representation of mixed + and · compositions
 - This allows **factorisation** and **exchange** to be made explicit, e.g.,
$$\text{Sys1} = M1 + M2$$
$$\text{Sys2} = (M1' + M2') \cdot M3$$
 - This means we keep refining a 'global plan' as well as refining the individual machines - at least until we reach a top-level product ·
- Shared event decomposition should cater for partitioning of external variables and events

Concluding

- Distinguishing problem structure from solution structure *feels right*
 - *Automotive features, distributed services, multicore operating systems, network services, ...*
- Working hypothesis:
 - Shared variables for problem composition
 - Shared events for solution composition

But alternatives should be explored too

- Factorisation and exchange support stepwise refactoring of structure from problem to solution
- Abstract algorithmic structures:
 - Need for rules for refactoring these

END