Incremental Design of Distributed Systems

Michael Butler
University of Southampton
August 2008

Alternative titles:

Applying Event-B and Incremental Refinement to Distributed Systems

Incremental Construction and Verification of *Models* of Distributed Systems

Building on Jean-Raymond's lectures

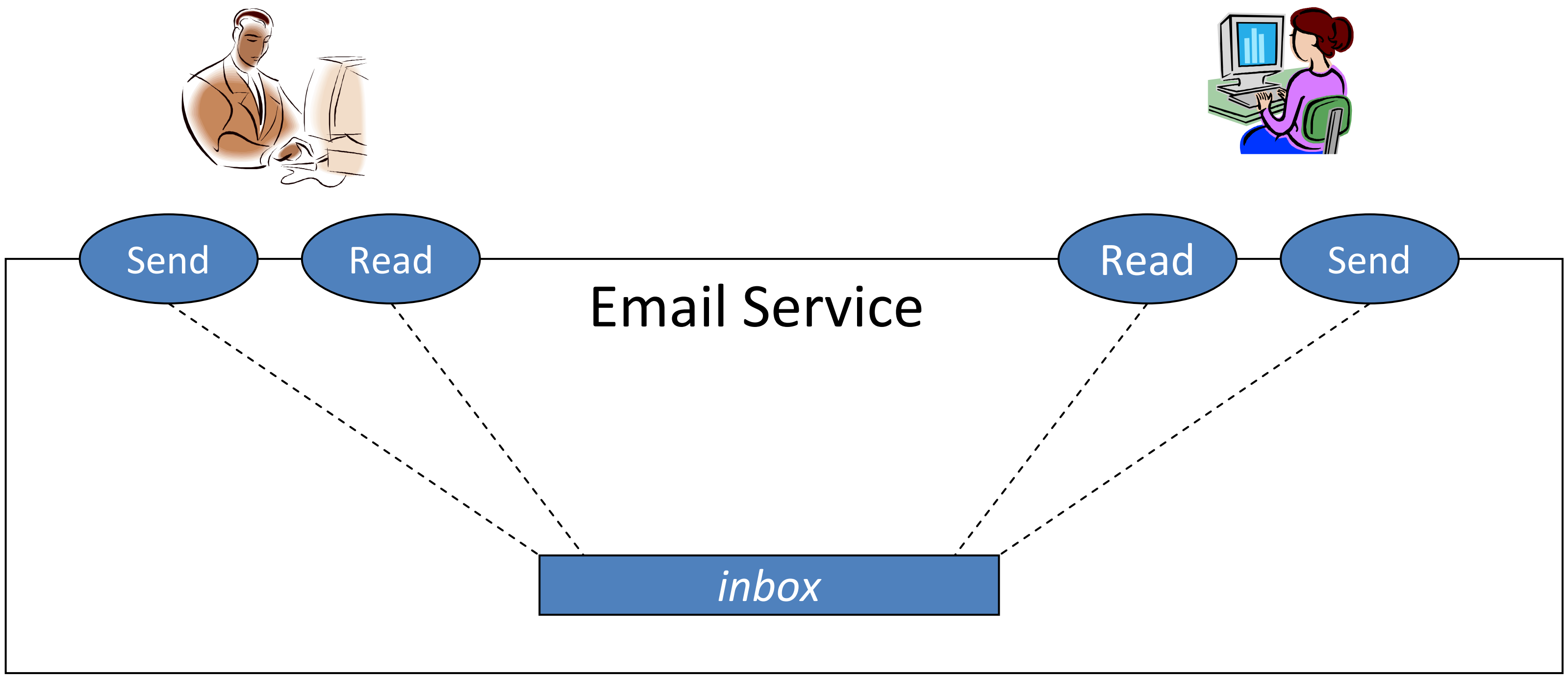
I will assume (some) knowledge of

- Event-B language
- Refinement
- Invariants
- Proof obligations
- Rodin tool

Key themes of my lectures

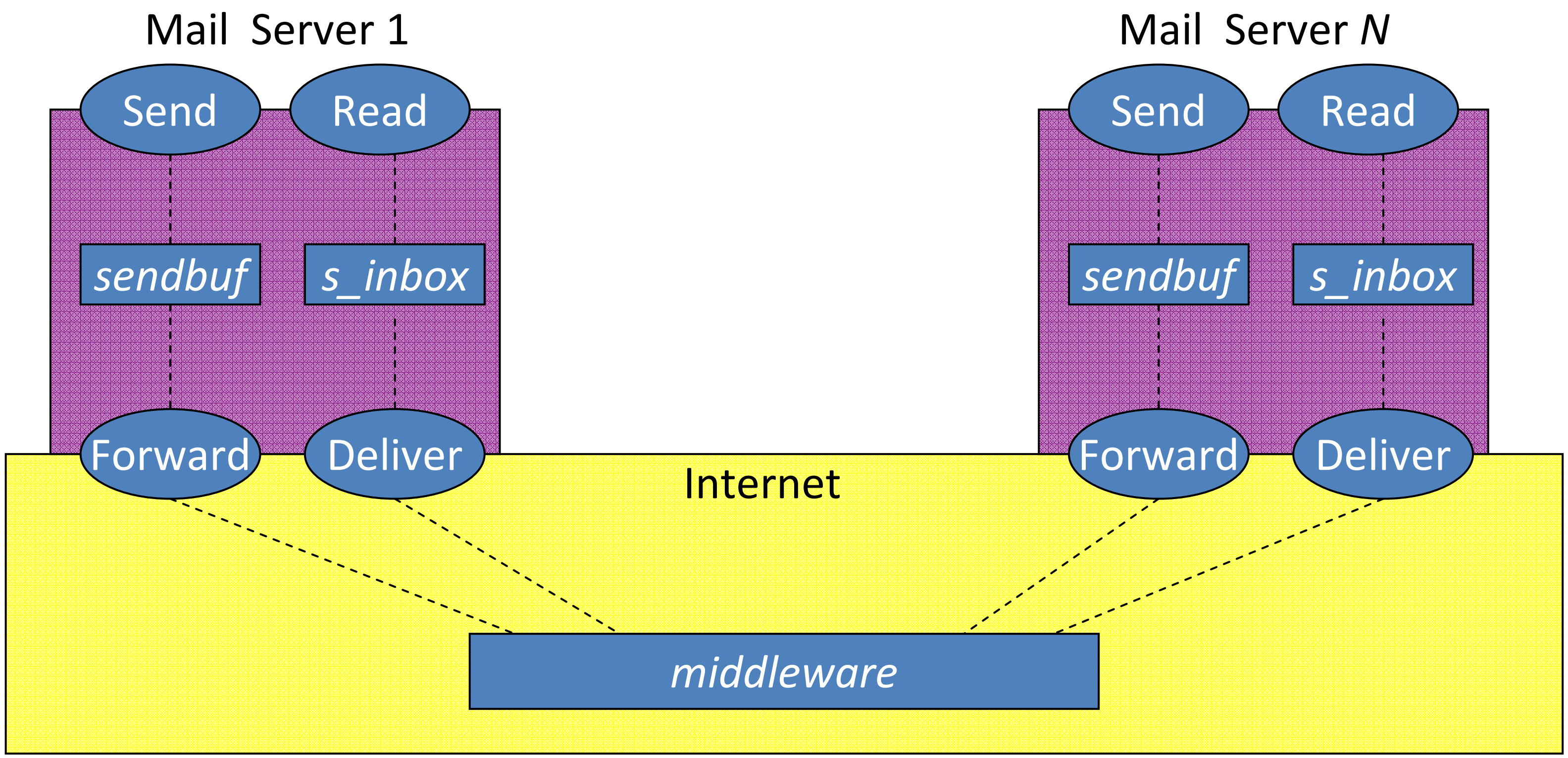
- Concurrency and atomicity in Event-B
- Atomicity refinement
  - Refining course-grained atomicity with more fine-grained atomicity
- Decomposing models into sub-models
- Distributed systems (in this context) :
  - Special case of concurrency where the only shared variables are buffers used for message-passing

Example: abstract model of email service



$inbox \in User \leftrightarrow Message$

Refine to servers and middleware



Data refinement: replace abstract *inbox* by *sendbuf, s_inbox, middleware*

Why *incremental m*odelling?

- Abstraction gap between *specification* and *implementation* is often too big for feasible reasoning (formal and informal)
- More effective to bridge the gap with a refinement chain of intermediate models
- Smaller abstraction gap means more automated proof
- More automated proof makes it easier to change models

Refinement is not top down!

- A completed refinement chain (or tree) is usually presented in a top-down manner.

- Construction of a refinement chain is rarely top-down
  - Requirements change
  - When proving $M1 \sqsubseteq M2$, it may be more convenient to find M3 such that

    $M1 \sqsubseteq M3$ and $M3 \sqsubseteq M2$
  - When proving $M1 \sqsubseteq M2$, we discover problems with M1
  - Our understanding of the system changes (improves) as we elaborate the design

Overview of lectures

- Introduction ✔
- Modelling atomicity and concurrency
  - behaviour traces
- Atomicity refinement
- Model composition and decomposition
- Incremental modeling of distributed systems
  - File transfer
  - Email service
  - Replicated database
  - Mondex

Atomicity and Concurrency

Simple concurrent program

```
processMain
varx : INT
begin
x := 0  ;
cobeginpin 1..Ndo//   fork then join of N
Inc(p)                          //   parallel processes
coend;
output(x)
end

processInc( p : 1..N )
begin
x := x+1                             //  atomic assignment
end
```

Simple concurrent program

```
processMain
varx : INT
begin
x := 0  ;
cobeginpin 1..Ndo
Inc(p)
coend;
output(x)
end

processInc( p : 1..N )
begin
x := x+1
end
```

What does this program achieve?

Why does it work?

How would we verify this?

Identify the atomic steps

```
processMain
varx : INT
begin
x := 0  ;
cobeginpin 1..Ndo
Inc(p)
coend;
output(x)
end

processInc( p : 1..N )
begin
x := x+1
end
```

Initalisex

Individual sub-process Inc(p) increments x exactly once

Output x after all sub-processes have completed

Event-B *model* with 3 events

- Initialisex
- Increment x
  – parameterised by process identifier p
- Output x

Event-B context for model of the concurrent program

context c1

sets PROC

constants N

axioms

axm2 : finite(PROC)

axm1 : N = card(PROC)

Variables of the model

machine M2

variables x, oInc, oOut

invariants

inv1 : $x \in \mathbb{N}$

inv2 : $oInc \subseteq PROC$ // set of processes for which the increment event has occurred

inv3 : $oOut \in BOOL$ // true when output event has occurred

initialisation $\triangleq$

act1 : $x := 0$

act2 : $oInc := \varnothing$

act3 : $oOut := FALSE$

Events of the model

Inc ≙

any p **where**

grd1 : $p \notin oInc$ // Inc has *not* occurred for process p

then

act1 : x := x+1

act2 : oInc := oInc ∪ {p}

end

Out ≙

any v! **where**

grd1 : oInc = PROC // Inc has occurred for *all* processes

grd2 : oOut = FALSE // Out event has not occurred

grd3 : v! = x // v! is an output parameter

then

act1 : oOut := TRUE

end

Event traces of the model

Assume   PROC = { p1, p2 }   N = 2

Event traces of the model are

⟨ Inc.p1, Inc.p2, Out.2 ⟩and ⟨ Inc.p2, Inc.p1, Out.2 ⟩

Trace is a sequence of *event labels*.

Event label consists of event name + parameter values

Event traces provide ***a*** definition of the observable behaviour of an Event-B model - *interleaving* semantics

Similar behavioural models are used in process algebra, e.g., CSP

Animation Demo

Abstract model of desired behaviour

machine M1

Out ≙ \\ Output the value N

any v! **where**

grd1 : oOut = FALSE

grd2 : v! = N

then

act1 : oOut := TRUE

end

Traces of M1: ⟨Out.N⟩

Relationship between traces

Consider a trace of M2: $\langle$ Inc.p1, Inc.p2, Out.2 $\rangle$

Use hiding to remove Inc events:

$\langle$ Inc.p1, Inc.p2, Out.2 $\rangle \backslash$ Inc = $\langle$ Out.2 $\rangle$

By treating *Inc* as a hidden event, traces of M2 look like traces of M1

Event hiding operator in CSP is defined in this way

Refinement proof in Rodin

Proof obligation for M1 $\sqsubseteq$ M2

N = card(PROC) // from context
oInc = PROC // guard of Out in M2
$\vdash$
x = N // output values are equal

What invariant could we use?
(Hint: x and oInc are variables)

Refinement proof in Rodin

Proof obligation for M1 $\sqsubseteq$ M2

N = card(PROC) // from context
oInc = PROC // guard of Out in M2
$\vdash$
x = N // output values are equal

Invariant: x = card(oInc)

Proof Demo

Some answers

- What does this program achieve?
  - output(N)

- Why does it work?
  - Invariant: x = card(oInc)

- How would we verify this?
  - Discharging refinement proof obligations

Verification helped us uncover *why* it works

Compare with Owicki-Gries method

- Owicki-Gries:
  - Rule for composing Hoare triples for each subprocess
  - Noninterference side-condition: process P1 must preserve any pre and post conditions of P2 (and vice versa)
  - Auxiliary variables required in example: oInc1 and oInc2

- Refinement
  - All preconditions and postconditions are encapsulated by a single invariant
  - All proof obligations become invariant preservation obligations, including the non-interference obligations
  - Set theory allows for a succint invariant: x = card(oInc)

Deterministic or nondeterministic?

```
process Main
var x : INT
begin
x := 0  ;
cobegin p in 1..N do
Inc(p)
coend;
output(x)
end

process Inc( p : 1..N )
begin
x := x+1
end
```

traces(M2) =
{ ⟨ Inc.p1, Inc.p2, Out.2 ⟩,
⟨ Inc.p2, Inc.p1, Out.2 ⟩ }

so M2 is nondeterministic

traces(M2) \ Inc =
{ ⟨ Out.2 ⟩ }

so we observe deterministic behaviour

Observations

- We refined a deterministic model by a non-deterministic model
  - We usually think of refinement as reducing non-determinism!
- Event-B modelling of simple concurrent was presented bottom-up!

Next lecture

- More detail on atomicity refinement

Lecture 2:
More on event refinement

Michael Butler

University of Southampton

August 2008

Simple concurrent program

```
process Main
var x : INT
begin
x := 0  ;
cobegin p in 1..N do
Inc(p)
coend;
output(x)
end

process Inc( p : 1..N )
begin
x := x+1
end
```

What does this program achieve?

Why does it work?

How would we verify this?

Some answers

- What does this program achieve?
  - output(N)
- Why does it work?
  - Invariant: x = card(oInc)
- How would we verify this?
  - Discharging refinement proof obligations

Verification helped us uncover *why* it works

Compare with Owicki-Gries method

- Owicki-Gries:
  - Rule for composing Hoare triples for each subprocess
  - Noninterference side-condition:
    - process P1 must preserve assertions used for P2 (and vice versa)
  - Auxiliary variables usually required: e.g., oInc1 and oInc2

- Refinement
  - All preconditions and postconditions are encapsulated by a single invariant
  - Proof obligations become invariant preservation obligations, including the non-interference obligations

Single invariant

- Merging assertions from Owicki-Gries for N=2:

$$\neg oInc1 \wedge \neg oInc2 \Rightarrow x=0$$
$$\neg oInc1 \wedge oInc2 \Rightarrow x=1$$
$$oInc1 \wedge \neg oInc2 \Rightarrow x=1$$
$$oInc1 \wedge oInc2 \Rightarrow x=2$$

  - Set theory allows for a succint invariant (for any N):

$$card(oInc) = x$$

Deterministic or nondeterministic?

```
process Main
var x : INT
begin
x := 0  ;
cobegin p in 1..N do
Inc(p)
coend;
output(x)
end

process Inc( p : 1..N )
begin
x := x+1
end
```

traces(M2)   =
{ ⟨ Inc.p1, Inc.p2, Out.2 ⟩,
⟨ Inc.p2, Inc.p1, Out.2 ⟩ }

so M2 is nondeterministic

traces(M2) \ Inc =
{ ⟨ Out.2 ⟩ }

so we observe deterministic behaviour by hiding Inc

Some important points

- Global invariants are easy to deal with when using set theory
- We refined a deterministic model by a non-deterministic model
  - Rrefinement is usually thought of as reducing non-determinism
- Event-B modelling of the simple concurrent program was presented bottom-up!

Event traces of a system M

Event labels Ev
States S
Initial states I
Labelled transition relation A $\in$ Ev $\longrightarrow$ (S $\leftrightarrow$ S)

Lift A to sequences AA $\in$ seq(Ev) $\longrightarrow$ (S $\leftrightarrow$ S) :

$$AA(\langle\rangle) = ID$$
$$AA(\langle e\rangle t) = A(e) \; ; \; AA(t)$$

AA(t)[I] is the set of states reachable by executing trace t

t $\in$ traces(M) **iff** AA(t)[I] $\neq \varnothing$

Note: traces are prefix-closed.

Event traces with hidden events

Transition relations

$A \in Ev \longrightarrow (S \leftrightarrow S)$

$H \in S \leftrightarrow S$

Lift AA:

$AA(\langle\rangle) = H^*$

$AA(\langle e\rangle t) = H^* ; A(e) ; H^* ; AA(t)$

$t \in traces(M)$ **iff** $AA(t)[\,I\,] \neq \varnothing$

Refinement

- M1 refined by M2
- Semantically: traces(M2)$\subseteq$traces(M1)
- Proof rule using gluing invariant J:

  Each M1.A_i is (data) refined by M2.A_i under J

  Each M2.H_i refines skip under J
- THEOREM: These are sufficient conditions for trace refinement

Simple file store example

sets FILE, PAGE, DATA

CONT = PAGE $\nrightarrow$ DATA

machine filestore
variables file, dsk
invariant
file $\subseteq$ FILE $\wedge$
dsk $\in$ file $\rightarrow$ CONT

initialisation
file := { } || dsk := { }

events

CreateFile = …

WriteFile = // set contents of *f* to be *c*
any f, c **where**
f $\in$ file
c $\in$ CONT
then
dsk(f) := c
end

ReadFile = // return data in page *p* of *f*
any f, p, d! **where**
f $\in$ file
p $\in$ dom(dsk(f))
d! = dsk(f)(p)
end

Sample event traces of file store

All prefixes of:

⟨ CreateFile.f1,
WriteFile.f1.c1,
ReadFile.f1.p3.c1(p3), … ⟩

All prefixes of:

⟨ CreateFile.f1,
CreateFile.f2,
WriteFile.f2.c4,
WriteFile.f1.c6, … ⟩

An (infinitely) many more traces.

Refinement of file store

- Instead of writing entire contents of a file in one atomic step, each page is written separately.

machine filestore2

refines filestore

variables file,dsk, writing,writebuf, sdsk

invariant

writing $\subseteq$ file

writebuf $\in$ writing $\to$ CONT

sdsk $\in$ writing $\to$ CONT // shadow disk

Refining the WriteFile event

- **Abstract**: WriteFile

- **Refinement**:

  StartWriteFile

  WritePage

  EndWriteFile (refines WriteFile)

  AbortWriteFile

Events of refinement

StartWriteFile=
any f, c **where**
f∈ (file \ writing)
c∈ CONT
then
writing := writing∪{f}
wbuf(f) := c
end

WritePage =
any f, p, d **where**
f∈ writing
p↦ d∈ wbuf(f)
then
sdsk(f) := sdsk(f) ∪ { p↦d}
end

Events of refinement

EndWriteFile
refines WriteFile
any f, c **where**
$f \in writing$
$c = sdsk(f)$
$dom(sdsk(f)) =$
$dom(wbuf(f))$
then
writing := writing \ { f }
wbuf := wbuf \ { f↦c}
dsk(f) := sdsk(f)
sdsk := sdsk\ { f↦ c }
end

AbortWriteFile
any f, c **where**
$f \in writing$
$c = sdsk(f)$
then
writing := writing \ { f }
wbuf := wbuf \ { f↦c}
sdsk := sdsk \ { f↦ c }
end

Comparing abstract and refined traces

⟨ CreateFile.f1,
CreateFile.f2,
WriteFile.f2.c2,
WriteFile.f1.c1
… ⟩

⟨ CreateFile.f1,
StartWriteFile.f1.c1,
CreateFile.f2,
WritePage.f1.p1.c1(p1),
StartWriteFile.f2.c2,
WritePage.f1.p2.c1(p2),
WritePage.f2.p1.c2(p1),
WritePage.f2.p2.c2(p2),
EndWriteFile.f2.c2,
WritePage.f1.p3.c1(p2),
EndWriteFile.f1.c1
… ⟩

Gluing invariant for file refinement

Gluing invariant

$\forall f \cdot f \in \text{writing} \;\Rightarrow\; \text{sdsk}(f) \subseteq \text{writebuf}(f)$

The *Rodin tool* was used to

- generate refinement obligations
- discharge the obligations
- guide the discovery of the invariant

Preserving liveness in refinement

- Enabledness preservation POs (not yet in Rodin tool):

  $$J \wedge grd(A) \Rightarrow$$
  $$grd(A') \vee grd(H_1) \vee \ldots \vee grd(H_n)$$

- Convergence POs using a variant V:
  each H_i decreases V

- **THEOREM**: Data refinement and liveness POs are sufficient for ***failure-divergence*** refinement (cf CSP)

Liveness POs for Owicki-Gries example

Enabledness :

grd(M1.Output) $\Rightarrow$

grd(M2.Output) $\vee$ grd(M2.Inc)

i.e.,

($\exists$ v! • oOut=FALSE $\wedge$ v!=N) $\Rightarrow$

($\exists$ v! • oOut=FALSE $\wedge$ oInc=PROC $\wedge$ v!=N)

($\exists$ p • p $\notin$ oInc)

Convergence:

M2.Inc decreases variant PROC \ oInc

Lecture 3

Michael Butler
University of Southampton
August 2008

Progress obligations in refinement

Enablednesspreservation POs (not yet in Rodin tool):

$J \wedge grd(\ M1.A\ ) \Rightarrow$
$grd(\ M2.A\ ) \vee grd(\ H_1\ ) \vee \ldots \vee grd(\ H_n\ )$

- Convergence POs using a variant V:
  each H_i decreases V

- **THEOREM**: Data refinement and liveness POs are sufficient for ***failures-divergence*** refinement (cf CSP)

Liveness POs for Owicki-Gries example

Enabledness :

grd(M1.Output) $\Rightarrow$

grd(M2.Output) $\vee$ grd(M2.Inc)

i.e.,

($\exists$ v! • oOut=FALSE $\wedge$ v!=N) $\Rightarrow$

($\exists$ v! • oOut=FALSE $\wedge$ oInc=PROC $\wedge$ v!=N)

($\exists$ p • p $\notin$ oInc)

Convergence:

M2.Inc decreases variant PROC \ oInc

References on failures-divergence treatment of action systems

Michael Butler

Stepwise Refinement of Communicating Systems

Science of Computer Programming, 27 (2), 1996

Michael Butler

A CSP Approach to Action Systems

PhD Thesis 1992

Some Event-B Experiments

- Replicated database
- Mondex electronic purse system

Replicated data base

- Abstract model

  $db \in object \rightarrow DATA$

Update = /* update a set of objects *os* */

any os,upd

where

$os \subseteq object \wedge$

$update \in (os \rightarrow DATA) \rightarrow (os \rightarrow DATA)$

then

db := db**<+**update(os◁db)

end

Refinement by replicated database

sdb $\in$ site $\longrightarrow$ (object $\longrightarrow$ DATA)

Update is by two phase commit:

- Global commit if all sites*pre-commit*
- Global abort if at least one site aborts

First refinement

- Introduce transaction identifiers
  - Each transaction has an object set and an update function on that object set
- Still use *db* (not yet *sdb*)

Events:

StartTrans(t) *refines skip*

AbortTrans(t) *refines skip*

CommitTrans(t) *refines Update*

Read(o,d!)*refines Read*

Second refinement

Replace *db* with *sdb*. Introduce 2 phases.

Events

StartTrans(t) *refines StartTrans*

PreCommit(t,s) *refines skip, locks objects used in t*

CommitTrans(t) *refines CommitTrans*

LocalCommit(t,s) *refines skip,*

updates sdb(s), releases objects

GlobalAbort(t) *refinesAbortTrans*

LocalAbort(t,s) *refines skip, releases objects*

Read(o,d) *refines Read guard: object is not locked*

Key gluing invariants

$$\forall s,o \cdot o \notin dom(lock(s)) \Rightarrow (sdb(s))(o) = db(o)$$

If an object is not locked at a site

then the value of the object at that site is the same as its value in the abstract global database

Key gluing invariants

$\forall t,s,o \cdot$
$t \in trans \ \land \ s \mapsto t \in precommit \ \land$
$t \notin commit \ \land \ t \mapsto o \in tos \ \Rightarrow$
$(sdb(s))(o) = db(o)$

If a transaction t
is in the precommit state at a site and
t has not yet globally committed and
o is an object of t
then the value of the object at that site is the same
as its value in the abstract global database

Key gluing invariants

$\forall t,s,o \cdot t \in commit \;\wedge$
$s \mapsto t \in precommit \;\wedge\; t \mapsto o \in tos \;\Rightarrow$
$\quad ((tupd(t))\ (tos[\{t\}] \lhd sdb(s)))\ (o) = db(o)$

If a transaction t

is in the precommit state at a site and

t has globally committed and

o is an object of t

then the value of the object at that site is found by applying the update associated with the transaction to the database at the local site.

Object contention

- Deadlock can occur when transactions require the same objects:
  - t1 locks o at site s1
  - t2 locks o at site s2

- Solutions
  - Abort a transaction when a required object is already locked
  - Use a global ordering on the transactions using Atomic Broadcasting primitives

- DivakarYadav. *Rigorous Design of Distributed Transactions*. PhD thesis, University of Southampton 2008.

Incremental development of Mondex in Event-B (with DivakarYadav)

- Constructed a refinement proof between
  - Abstract model of system of purses including balance transfer, loss, recovery and balance check
  - Detailed model of distributed system of purses including abort, archiving, messaging
- Very high degree of automatic proof (B4Free tool)
- Refinement chain with 10 levels
  - Small abstraction gap at each stage – simpler invariants
  - **Not** top down

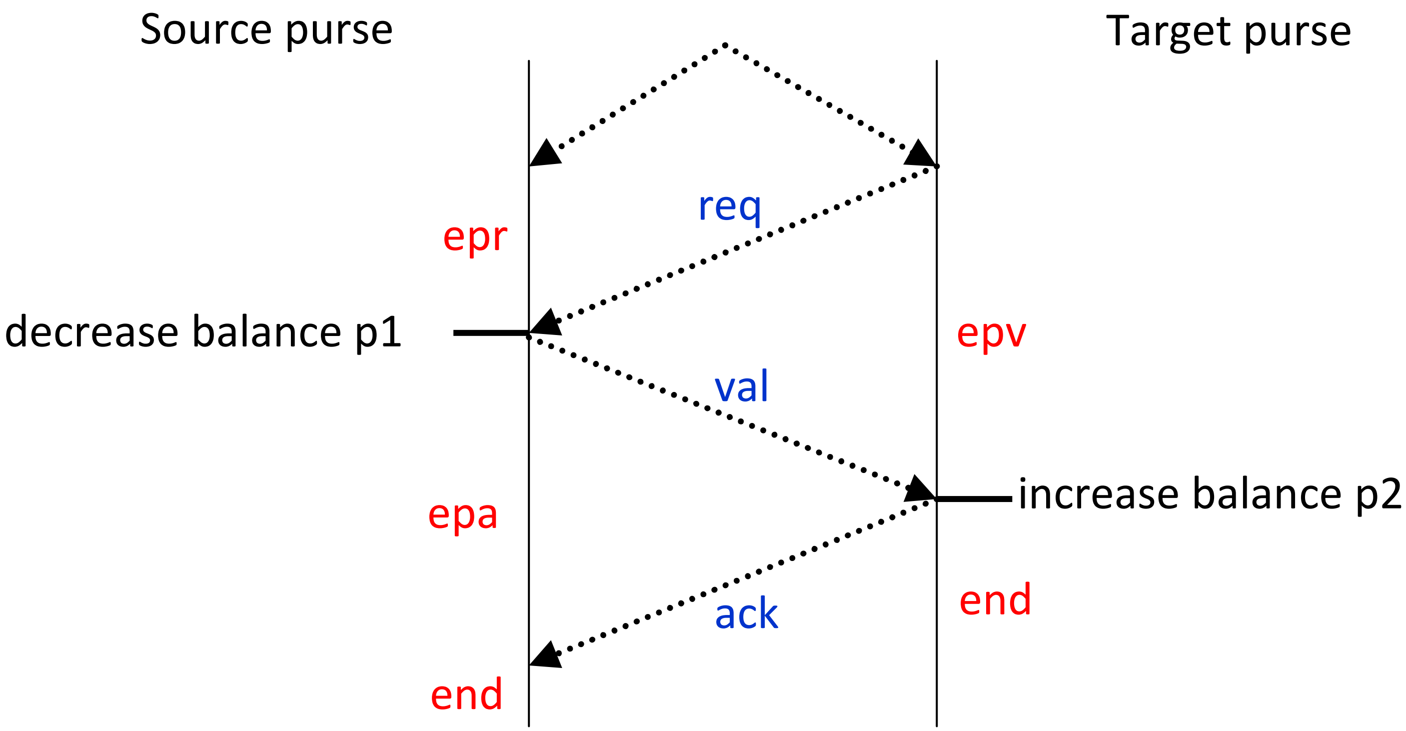
Abstract spec of Mondex purses

TransferOk =
whenbal(p1) $\geq$ a **then**
bal(p1) := bal(p1)-a || bal(p2) := bal(p2)+a **end**

LoseValue =
when bal(p1) $\geq$ a **then**
bal(p1) := bal(p1)-a || lost(p1) := lost(p1)+a **end**

Recover =
when lost(p1) $\geq$ a **then**
bal(p1) := bal(p1)+a || lost(p1) := lost(p1)-a **end**

Protocol steps



Also: a transaction can be aborted at any point
Abort caused by timeout or by card removal

Intermediate abstraction

- Abstraction gap is too big
- Introduce *transactions*:
  - Uniquely identified
  - Have attributes (source, target, amount)
  - Have abstract end-to-end state:
    pending, *ended*, *recoverable*
  - *pending*: *val* is in transit
  - *recoverable*: amount has been added to *lost*

Overview of refinement chain

- L1: Atomic transfer of value and recovery of lost value
- L2:Transactions introduced with end-to-end state.
  - Balance transfer split into 2 events
  - Freshness of new transactions based on history
- L3: Remove some redundancy
- L4: End-to-end state replaced by dual state

  (epr, epv, abortepv, …)
- L5: Explicit messaging between terminal and purses and between purses

Overview of refinement chain

- L6: Introduce for each purse
  - 1 current trans + archive of aborted trans
- L7: Remove global history
  - Freshness ensured by individual purses with fresh transaction numbers
- L8: Make fresh purse number sequential
- L9: Change representation of messages to a record structure
- L10: Change representation of transaction states from disjoint sets to state function

Guideline

- Use separate disjoint sets instead of single function to represent the discrete control states
- Good:

  pending $\subseteq$ Transaction

  recoverable $\subseteq$ Transaction

  ended $\subseteq$ Transaction

  disjoint (pending, recoverable, ended)

  Get quantifier-free gluing invariant:

$abortepas \cap abortepvs \subseteq recover$

- Not so good:

  status $\in trans \rightarrow Status$

- Function form can be introduced later as a refinement (which is provable completely automatically)

Proof statistics with B4Free tool

Level	POs	Interactive
L1	24	0
L2	91	15 (av 10 steps) (sum, finiteness)
L3	14	0
L4	143	0 (end-to-end)
L5	57	0 (messaging)
L6	183	0 (localise to purses)
L7	25	0
L8	23	2 (av 5 steps)
L9	73	0
L10	46	0
totals	679	17

97.5% of POs proved fully automatically

Refinement of Recovery

Abstract:

when lost(p1) $\geq$ a **then**

bal(p1) := bal(p1)+a ||

lost(p1) := lost(p1)-a

end

Concrete:

when

t$\in$ archive(p1) $\wedge$t$\in$ archive(p2)

p1 = from(t)$\wedge$p2 = to(t) $\wedge$ a = am(t)

then

cbal(p1) := cbal(p1)+a

archive(p1) := archive(p1) \ {t}

archive(p2) := archive(p2) \ {t}

end

Observation: importance of global reasoning

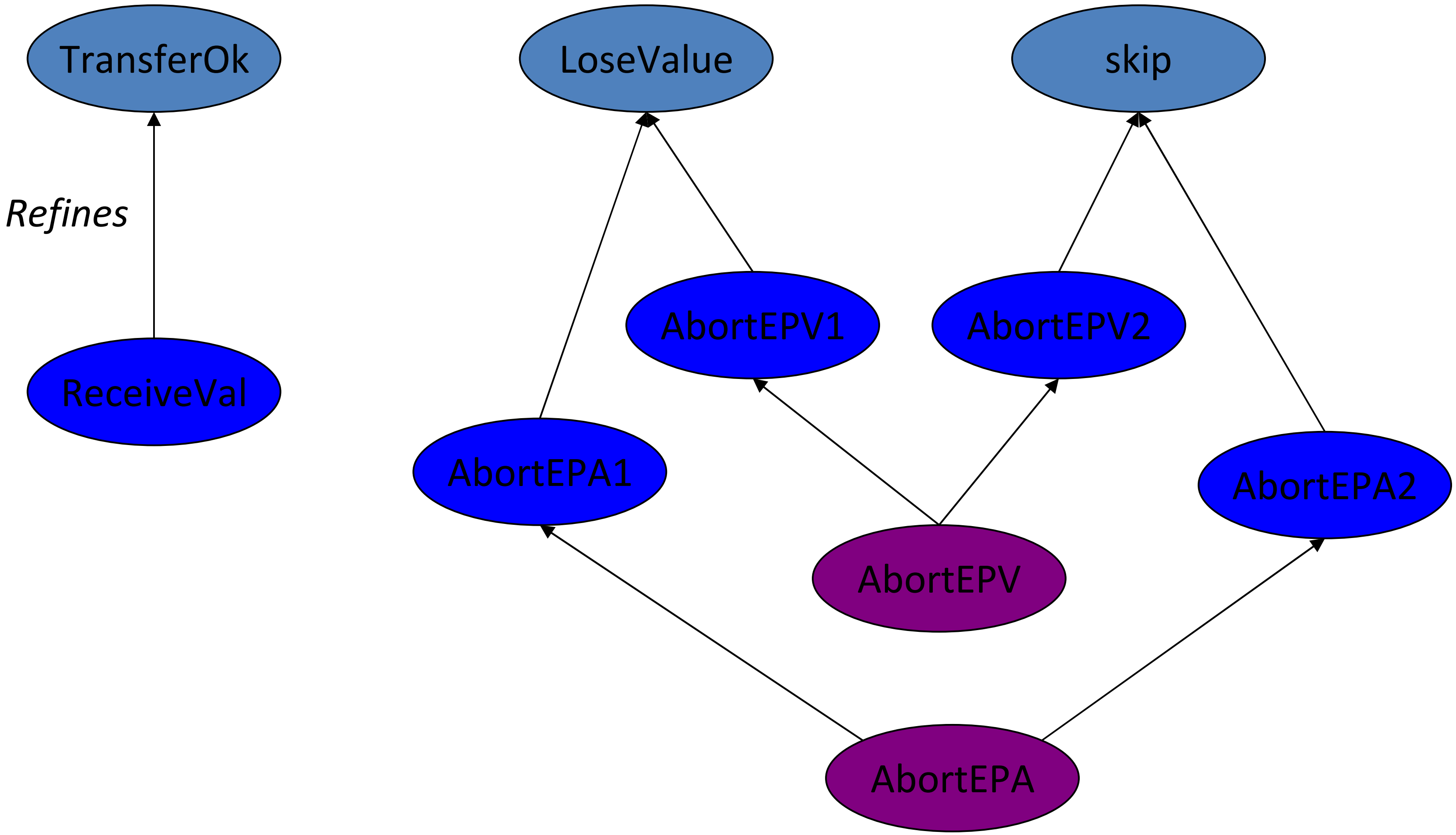
- Two cases for p2 aborting in epv state:
  - AbortEPV1: p1 has already aborted
  - AbortEPV2: p1 has not aborted

  The distinction cannot be made locally

- AbortEPV1 refines LoseValue
- AbortEPV2 refines skip

- Similarly 2 cases for AbortEPA

Multiway Refinement in Mondex



Balance Check

- Abstract:

ExactBalanceCheck(p)   b! = bal(p)

InexactBalanceCheck(p)   b! ≤ bal(p)

- Concrete:

ExactBalanceCheck(p) is guarded by

    - p not involved in a transaction and
    - p has no outstanding aborted transactions

InexactBalanceCheck(p) is guarded by negation of these conditions

Overview of effort

- Approx 2 weeks devoted to modelling and proof (but longer elapse time)
- Interactive proof was mostly used for discovering invariants and fine tuning of models
- Most invariants were discovered by inspecting unproved POs
- Re-enforced key guidelines for minimising proof effort

Guideline

- Keep data ***As Abstract As Possible*** when introducing algorithmic / distributed / fined-grained structure

- Example: intermediate end-to-end state for transactions made it easy to express the invariant required to break the atomicity of the abstract transfer events

Lecture 4
Decomposition of Models

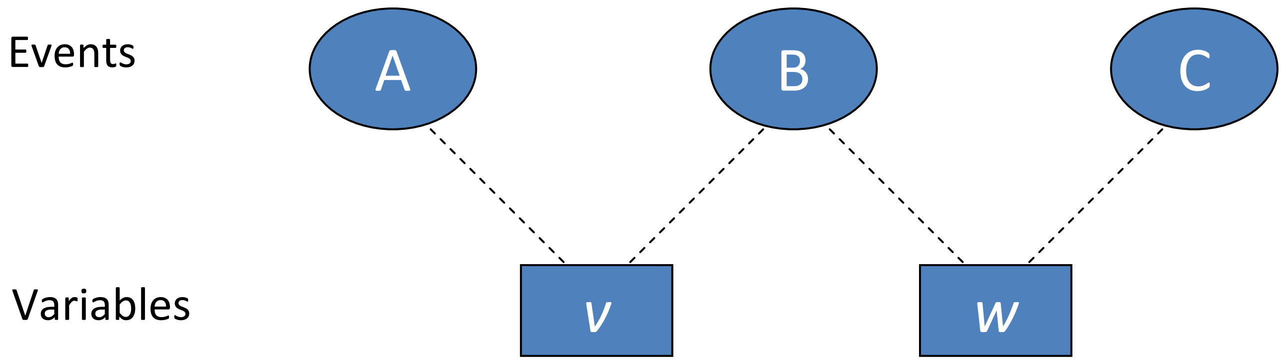
Michael Butler

University of Southampton
August 2008

Decomposition

- When models become too big we need to decompose them
- Decomposition also reflects architectural structure
- Approach: we define a (parallel) composition operator on Event-B machines
  - M1 || M2
- Decomposition: refine model to a sufficient degree that the composition operator can be applied (in reverse)
  - $M \sqsubseteq M1 \,||\, M2$
- M1 and M2 can be further refined and decomposed

Decomposition – by example

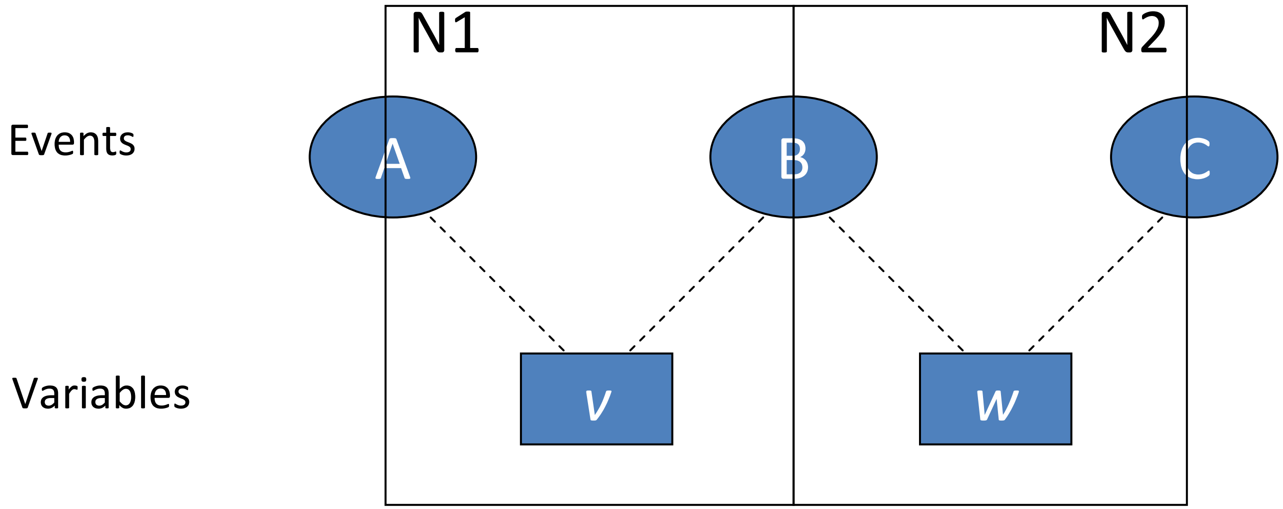


A = v := v+1

B = **when** v>0 ∧w<M **then** v := v-1 || w := w+1 **end**

C = **when**w>0 **then**w := w-1 **end**

Decompose by partitioning variables

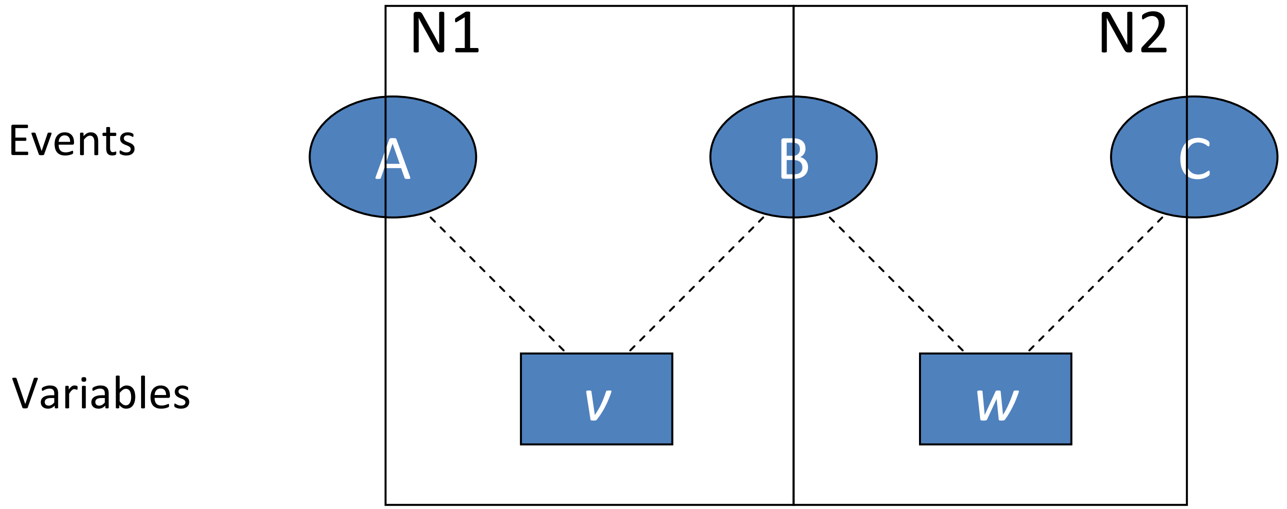


A = v := v+1

B = **when** v>0 ∧w<M **then** v := v-1 || w := w+1 **end**

C = **when**w>0 **then**w := w-1 **end**

Parallel Event Split



B = **when** v>0∧w<M **then** v := v-1 || w := w+1 **end**

B is split into two parallel events operating on independent vars:

B_1 = **when** v>0 **then** v := v-1 **end**

B_2 = **when** w<M **then** w := w+1 **end**

Synchronised events with parameter passing

B = **any** x **where** $0 < x \leq v$

then $v := v - x$ || $w := w + x$ **end**

B can be split into 2 events that have x in common:

B_1 = **any** x **where** $0 < x \leq v$ **then** $v := v - x$ **end**

B_2 = **any** x **where** $x \in \mathbb{N}$ **then** $w := w + x$ **end**

B_1 constrains the value for *x* by $0 < x \leq v$ (output)

B_2 just constrains the value of *x* to a type (input)

Synchronised Composition Operator

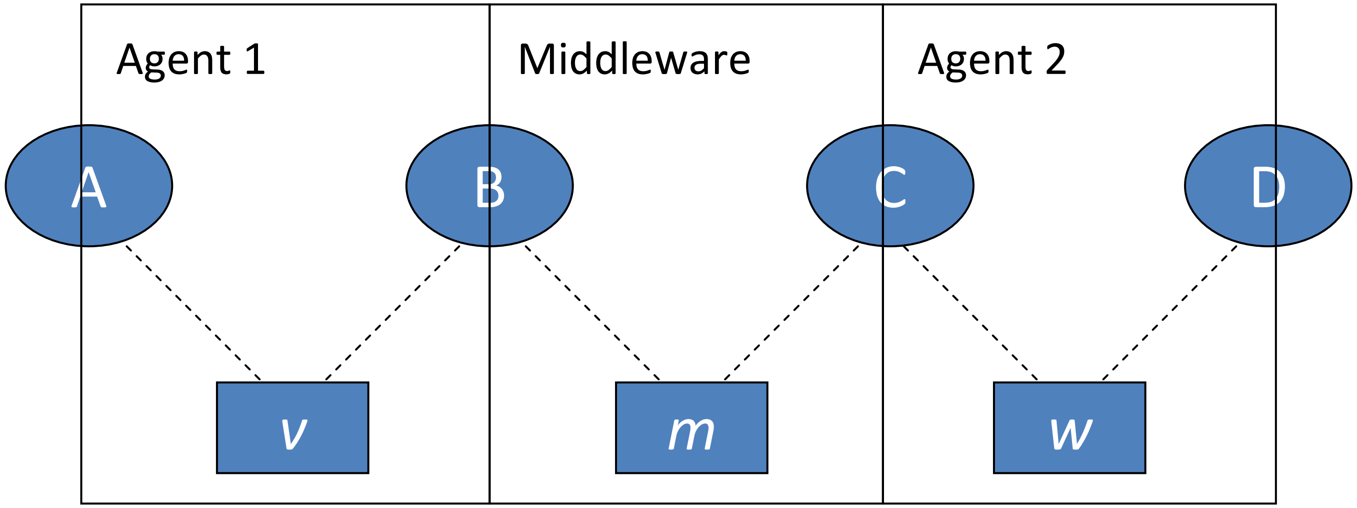
- Synchronised composition operator for Event-B machines is syntactic
  - combine guards and combine actions of events to be synchronised
  - no shared state variables
  - common event parameters represent values to be agreed on synchronisation by both parties
- Corresponds to parallel composition in CSP
  - process interact via synchronised channels
  - monotonic: subsystems can be refined independently!

Composition Plug-in for Rodin

composed machine M2
refines M1
includes N1, N2
events
A = N1.A
B = N1.B || N2.C
C = N3.C
end

Tool generates POs to verify that M2 is refined by the composition of N1 and N2

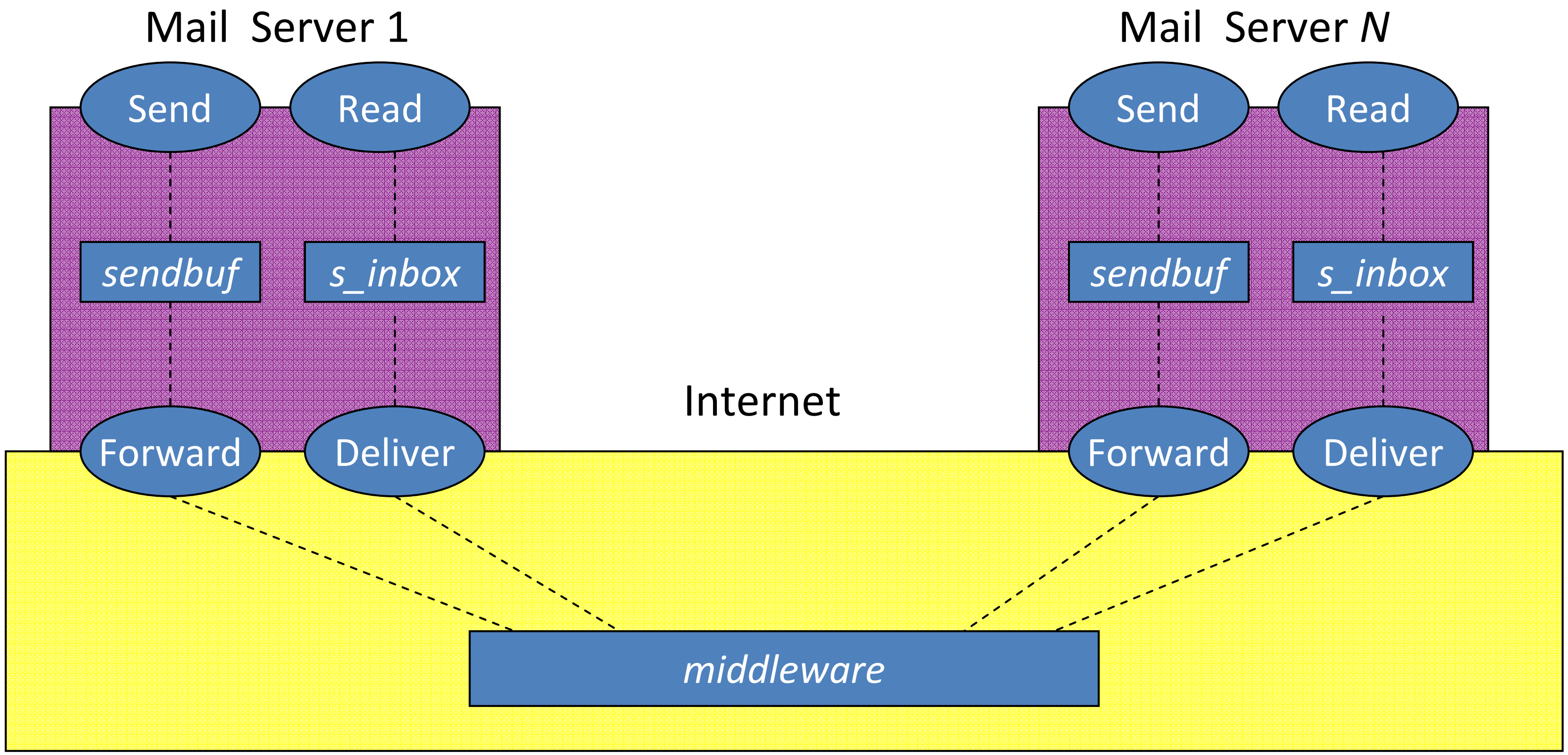
Asynchronous distributed system



For distributed systems, agents do not interact directly.

Instead they interact via some middleware, e.g., the Internet

Decomposition of mail service



References on synchronised composition of action systems

Michael Butler

Stepwise Refinement of Communicating Systems

Science of Computer Programming, 27 (2), 1996

Michael Butler

A CSP Approach to Action Systems

PhD Thesis 1992

Observation on Decomposition

- Typical approach: refine M by N1 || N2
- The decomposition itself is easy
  - Essentially a syntactic decomposition
- The more challenging part is refining the abstract “global” model to a sufficiently detailed model to allow the syntactic decomposition to take place

Simple file transfer

invariants

inv1 : $fileA \in PAGE \nrightarrow DATA$

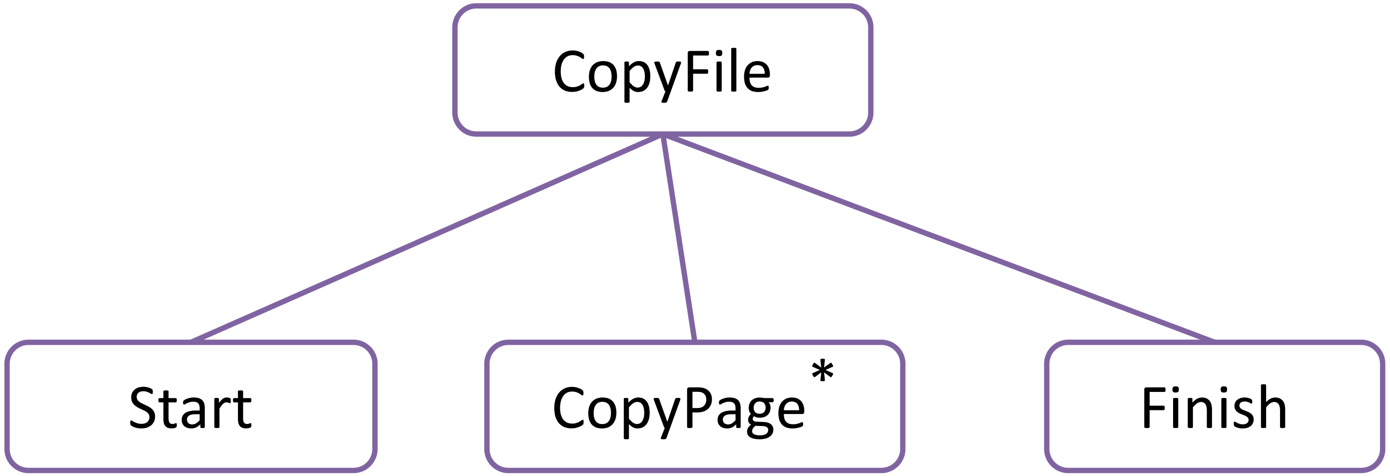
inv2 : $fileB \in PAGE \nrightarrow DATA$

events

CopyFile $\triangleq$ $fileB := fileA$

end

Diagrammatic representation of event refinement



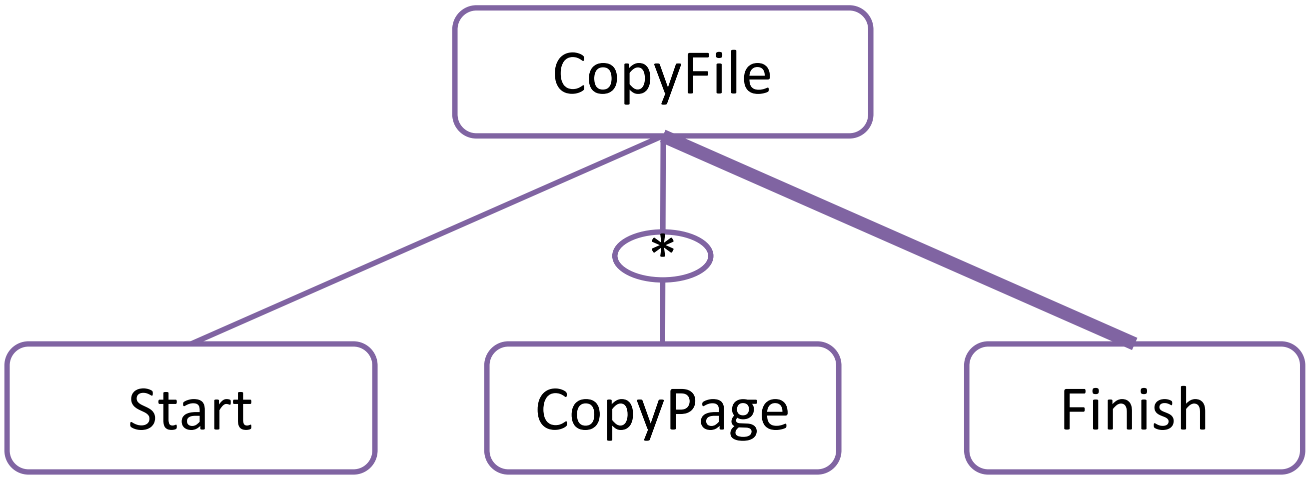
Sequencing is from left to right

* signifies iteration

Jackson Structure Diagrams

- Part of Michael Jackson's Structured Development Method JSD
- Graphical representation of behaviour
- We can exploit the hierarchical nature of JSD diagrams to *illustrate* event refinement
- Adapt JSD notation for our needs

Adapting the diagrams



Events are represented by leaves of the tree only
Attach the operator to an arc rather than a node to clarify atomicity
Heavy line indicates *Finish* refines *CopyFile*

NB: This is not a class diagram. It describes behaviour.

First refinement

invariants

inv1 : $buf \in PAGE \twoheadrightarrow DATA$

inv2 : $oStart=TRUE \Rightarrow buf \subseteq fileA$

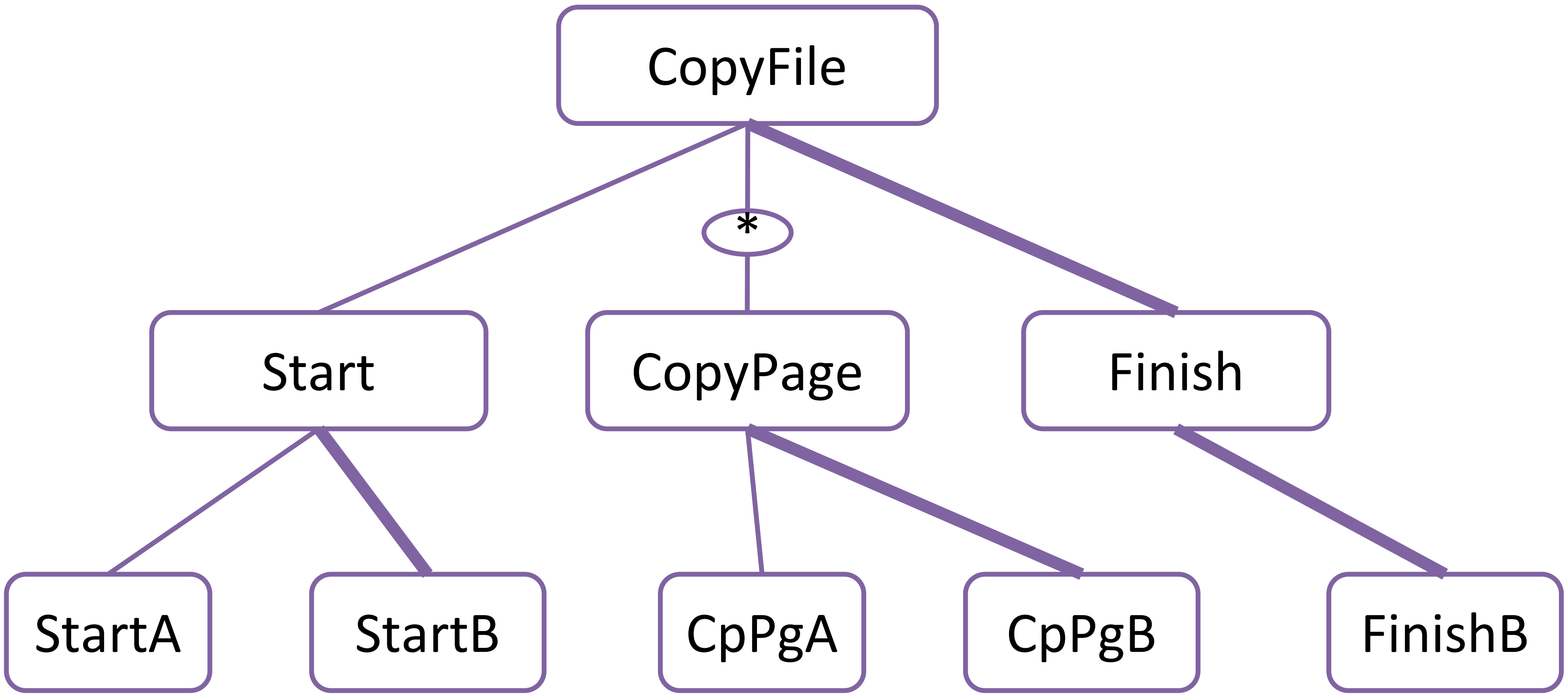
events

Start ≙ **...** oStart := TRUE || buf := ∅**...**

CopyPage ≙
any p, d**where**
oStart = TRUE
$(p \mapsto d) \in fileA \setminus buf$
then
buf := $buf \cup \{ p \mapsto d \}$
end

Finish ≙
refines CopyFile
when
oStart = TRUE
card(buf) = card(fileA)
oFinish = FALSE
then
fileB := buf
oFinish := TRUE
end

Further event refinement: introduce more asynchrony



First refinement

variables

bufA , bufB, oStartA, oStartB, oEnd

events

StartA≙ … oStartA := TRUE …

StartA≙ … oStartB := TRUE …

CpPgA ≙ … bufA := bufA ∪ { p↦d } …

CpPgB ≙ … bufB := bufB ∪ { p↦d } …

End ≙ … fileB := bufB…

Strong dependency between A and B

StartA ≙

when

oStartA = FALSE

then

oStartA := TRUE

bufA := ∅

sizeA := card(fileA)

end

StartB ≙

when

oStartA= TRUE

oStartB = FALSE

then

oStartB := TRUE

sizeB := sizeA

bufB := ∅

end

StartB event can read variables belonging to A side

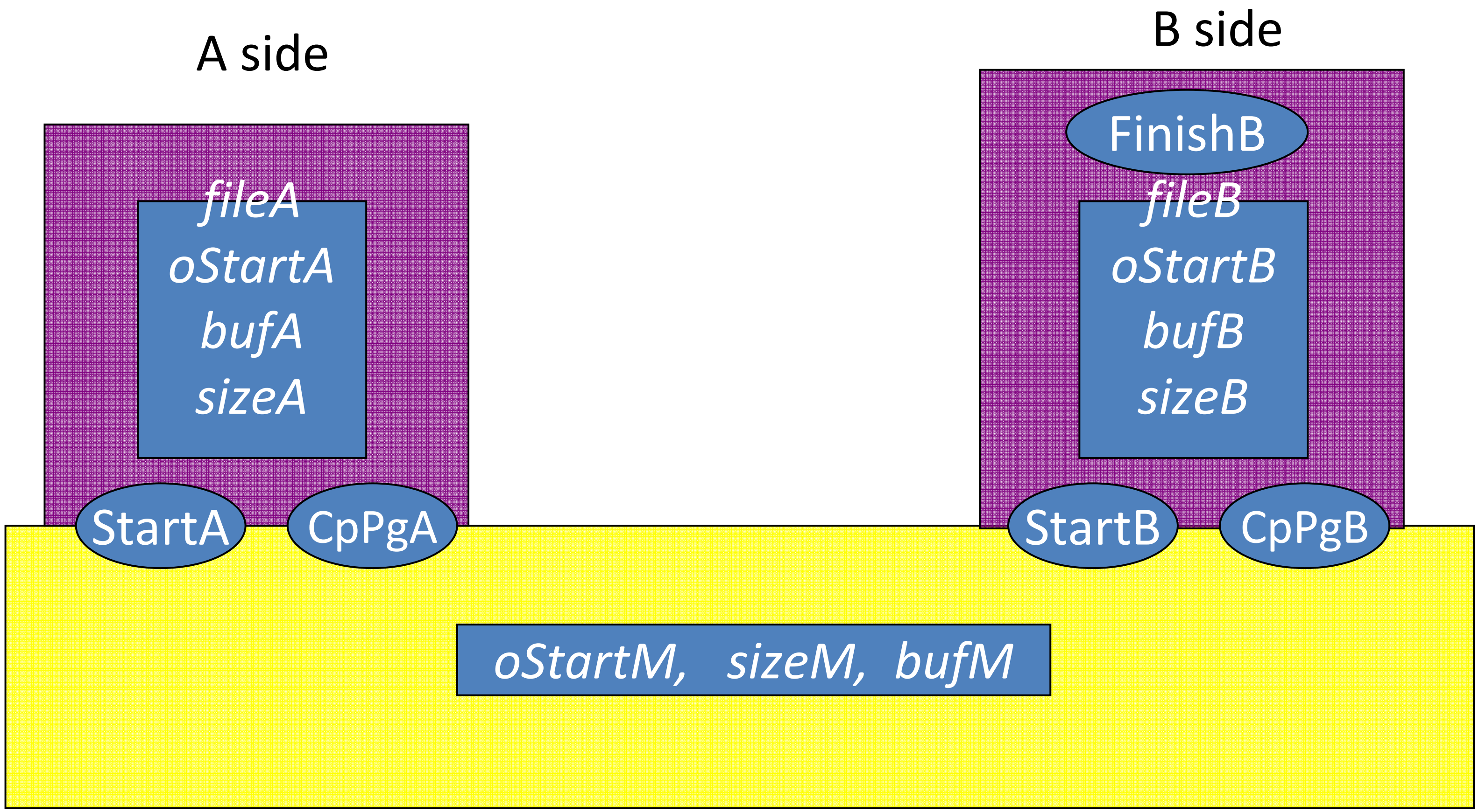
Weaken the dependency: introduce shared buffer through refinement

StartA ≙
when
oStartA = FALSE
then
oStartA := TRUE
oStartM := TRUE
bufA := ∅
sizeA := card(fileA)
sizeM := card(fileA)
end

StartB ≙
when
oStartM= TRUE
oStartB = FALSE
then
oStartB := TRUE
sizeB := sizeM
bufB := ∅
end

invariant: oStartM=oStartA, sizeM=sizeA

Decomposition of file transfer

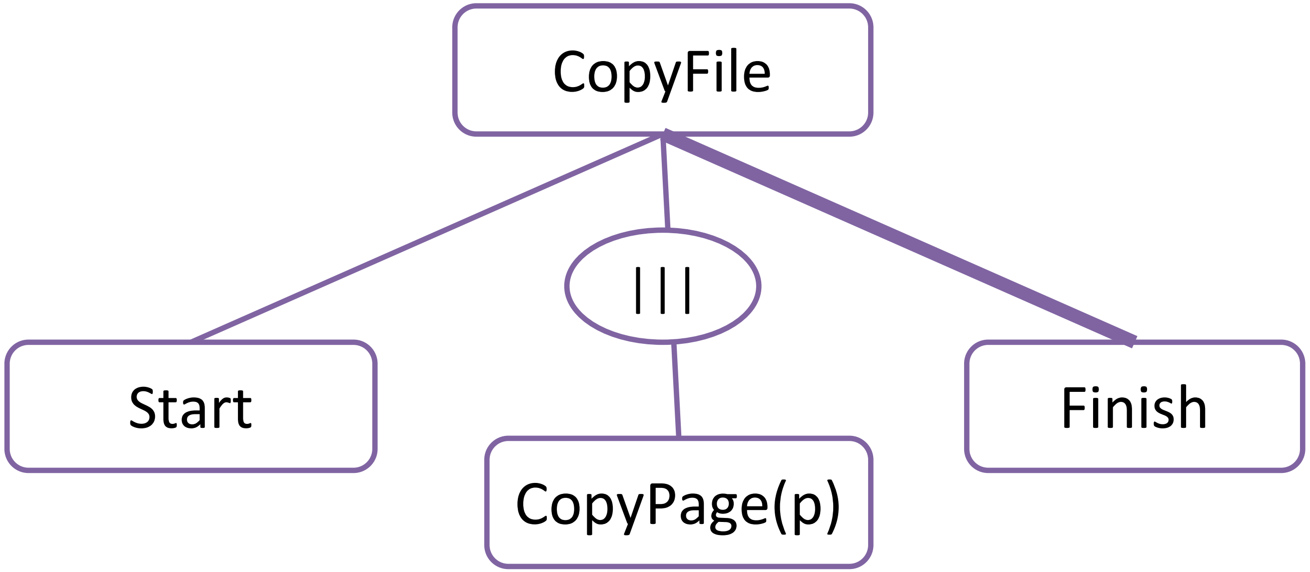


Further refinement

All 3 components can now be refined independently:

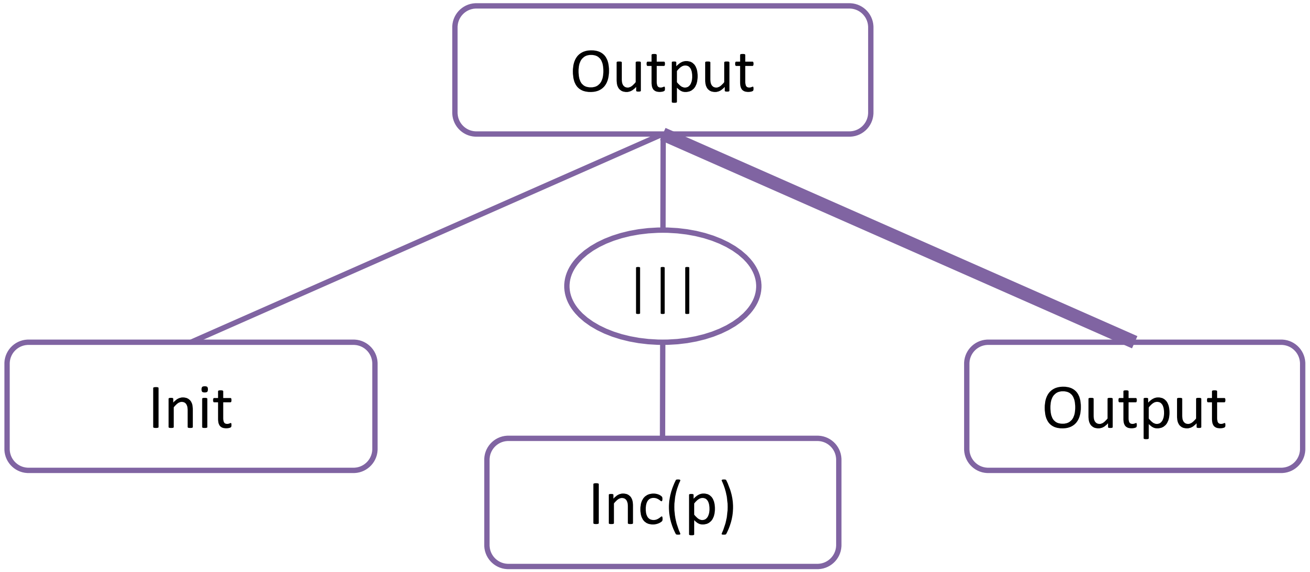
- Data structures of SideA and SideB can be optimised
- Middleware can be fined by introducing a more explicit representation of messages as variant records (or classes and subclasses)
  - Init message contains file size
  - Step message contains a page of data

Aside: interleaving instead of sequencing



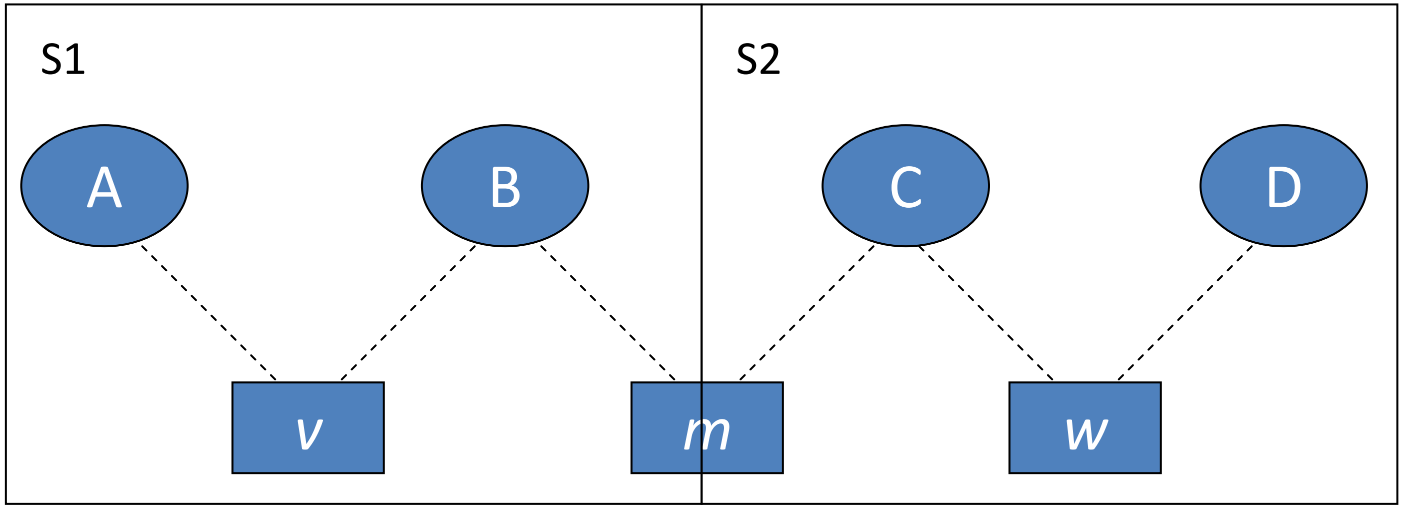
Illustrates multiple interleaved instances of CopyPage event
Each instance is identified by $p \in PAGE$

We have seen this pattern already



Here $p \in PROC$

Alternative style of decomposition

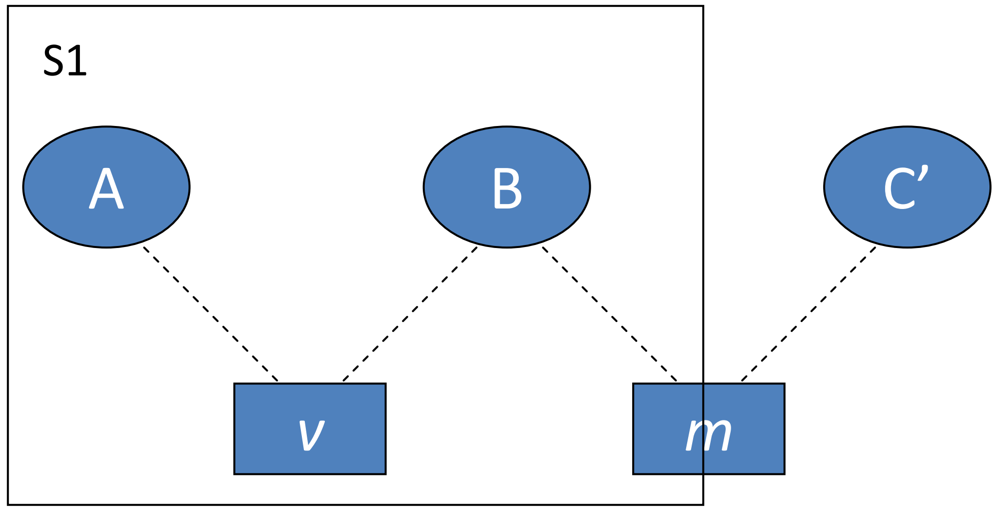


Events are independent

S1 and S2 interact through shared variables

S1 and S2 need to be refined in a consistent way

Environment obligation in refinement



C’ must maintain any invariants used to refine S1

For composition, environment events that modify m must refine C’

Abrial and Hallerstede.
Refinement, decomposition and instantiation of discrete models.
FundamentaeInformatica, 2006.

Tomorrow

- Other Event-B tools
  - UML-B
  - ProB
- Future plans for Rodin toolset

Lecture 5

Michael Butler

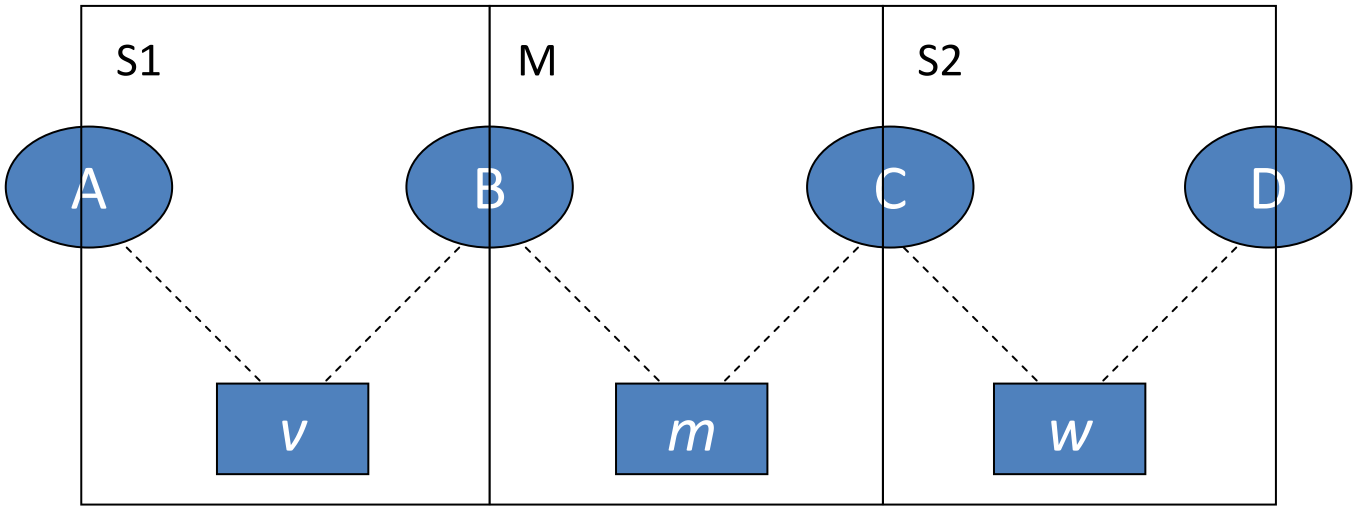
University of Southampton

August 2008

Today

- (A little) more on decomposition
- Security: model of access control example
  - Rodin demo
- ProB
- UML-B
- Future plan for Rodin

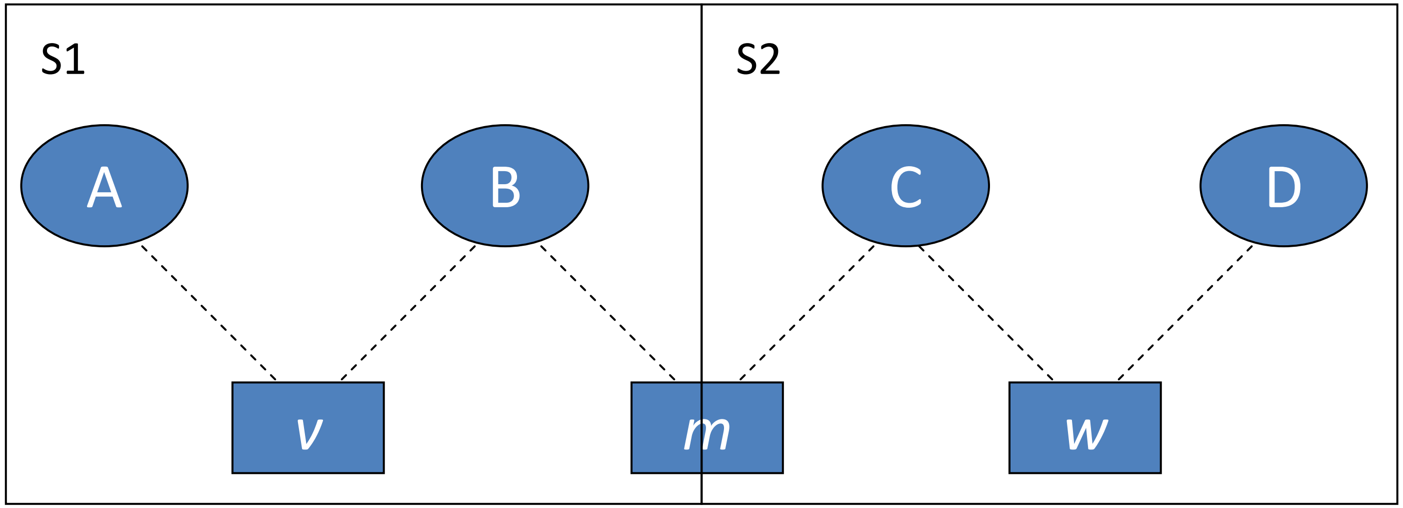
Synchronised composition of machines



Variables are partitioned

B and C are synchronised events

Alternative style of decomposition

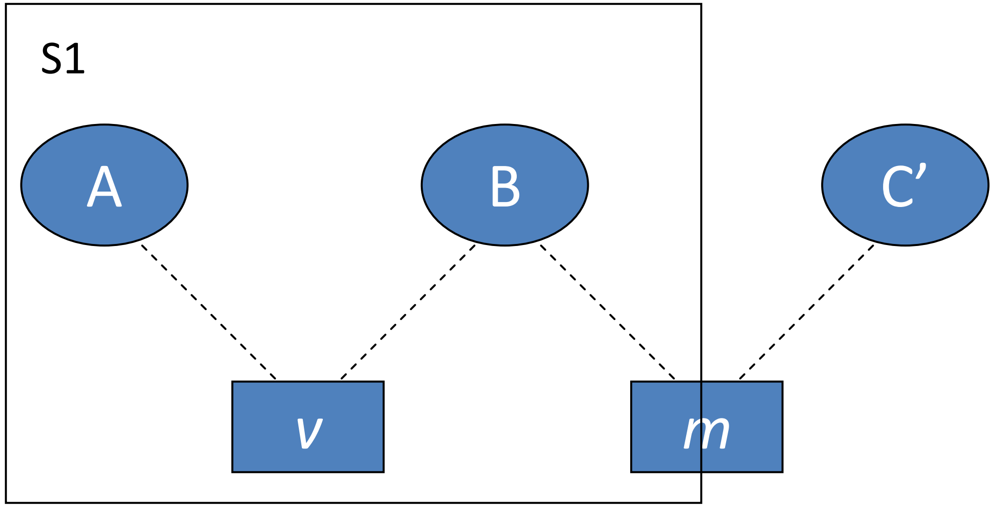


Events are partitioned

Variable m is shared by S1 and S2

S1 and S2 need to be refined in a consistent way

Environment obligation in refinement



C’ must maintain any invariants used to refine S1

For composition, environment events that modify m must refine C’

Abrial and Hallerstede.
Refinement, decomposition and instantiation of discrete models.
FundamentaeInformatica, 2006.

Demo: Access Control System

- Users are authorised to engage in activities
- Activities take place in rooms
- Users can only be in a room if they have authorised to engage in *each* activity that takes place in that room

Rodin Implementation

- Extension of Eclipse IDE (Java based)
- Repository of modelling elements (Java objects, XML files)
- Rodin Eclipse Builder manages:
  - Well-formedness + type checker
  - Consistency/refinement PO generator
  - Proof manager
  - Propagation of changes

Rodin Implementation

- Extension of Eclipse IDE (Java based)

- Rodin core development team:
  - Laurent Voisin (Systerel)
  - Stefan Hallerstede (Southampton)
  - Farhad Mehta (ETH)
  - Thai Son Hoang (ETH)
  - Francois Terrier (ETH)

www.event-b.org

Key Tool Decisions

- Support incremental development
  - *Reactive*: analysis tools are automatically invoked in the background whenever a change is made
  - *Differential*: analytical impact of changes is minimised as much as possible
  - Support strong interplay between modelling and proof – model can be changed during a proof
- Extensibility support:
  - extend modelling elements
  - extend functionality through plugins

Rodin Plug-ins

- Linking UML and Event-B
  - Colin Snook + Butler (Southampton)
- ProB: consistency and refinement checking
  - Michael Leuschel + team (Düsseldorf)
- Graphical model animation
  - Brama (Clearsy)
  - AnimB (Christophe Metayer)

ProB

- Animator and model checker
  - searches for invariant violations
- Originally developed for “Classical” B
- Now being ported to Event-B and Rodin
- Implementation uses symbolic representation using constraint logic programming
  - makes all types finite
  - exploits symmetries inB types

UML-B

- UML-like language
- Package, Class, State diagrams
  - Package used to structure a refinement chain
  - Class represents a set
  - Attributes and associations represent relations
  - UML-B classes have events
- Event-B as constraint and action language
  - Guards, invariants, actions
- UML-B plug-in for Rodin
  - Generates Event-B from UML-B

Demos

- ProB
- UML-B

Future

- Rodin coordination
- Deploy project

Rodin Coordination Committee

- Role: Ensure the coordinated evolution of the Rodin platform at a strategic level

- Current members
  - Michael Butler (Chair)
  - Jean-Raymond Abrial
  - Cliff Jones
  - Stefan Hallersede
  - TherryLecomte
  - Michael Leuschel
  - Laurent Voisin

DEPLOY Project (EU2008-2012)

- Aim: industrial deployment of formal engineering methods for high productivity and dependability

- 12 Partners:
  - Bosch, Siemens, SAP, Space Systems Finland
  - Systerel, CETIC, ClearSy
  - Universites: Newcastle, ÅboAkademi, ETH Zurich, Düsseldorf, Southampton

www.deploy-project.eu

Future

- Mathematical language extension support
- Links with other provers (FO, SAT, SMT, HO)
- User interface improvements (text editor)
- Requirements management and traceability
- Documentation management
- Model decomposition management
- Refinement pattern management
- UML-B: improve support for refinement
- Code generation from Event-B
- Proof cross checking
- …

Rodin

- RODIN platform supports incremental development of proved model chains in Event-B
- Architecture makes it possible to extend the language and the set of analysis tools
- Open source and will continue to be developed through EU project: www.deploy-project.eu

www.event-b.org