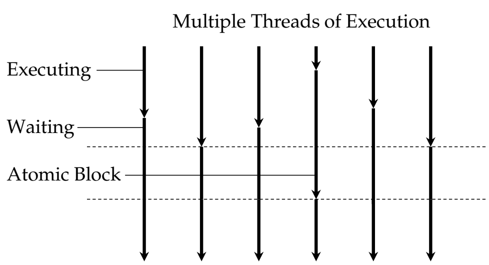
Synchronization Systems that Permit the Use of Large Atomic Blocks

Paul Sivilotti
Computer and Information Science
Ohio State University
paolo@cis.ohio-state.edu

Outline

- Utility of implicit atomic blocks.
- Background and related work.
- Main result: theorem of equivalence.
  - three refinement conditions
- Application of result to current systems.
- Implications for design of synchronization systems.

Atomic Blocks



Atomic Blocks

- *Atomic block*: execution is not interleaved with other actions.
- Atomic blocks are useful:
  - critical sections requiring mutual exclusion
  - simplify reasoning about the system
- One implementation: acquire/release locks.

```
lock.acquire();
 //atomic block
lock.release();
```

- Expensive in large systems.

Implicit Atomic Blocks

- Some blocks *appear* to be atomic.
  - *e.g.*, a block that manipulates local data only (no sends or receives to other processes)
  - intermediate states are not externally visible
- It is impossible to distinguish the case where such a block executes atomically from those where it does not.

Example: Program Simple

```
p1:

    ? x
    int a = x + 1
    ! a
    int b = a * a
    ! b
```

```
p2:

    int y = 5
    ! y

    ? y1
    ? y2
```

Example: Program Simple$_{atomic}$

```
p1_atomic:
<
    ? x
    int a = x + 1
    ! a
    int b = a * a
    ! b
>
```

```
p2_atomic:
<
    int y = 5
    ! y
>
    ? y1
    ? y2
```

Informal Rule

- "A receive action, followed by any number of send and/or internal actions, can be treated as an atomic block of code."

```
p:
    receive action;
    send and local actions;
```

Informal Rule

- "A receive action, followed by any number of send and/or internal actions, can be treated as an atomic block of code."

```
p:
    receive action;
    send and local actions;
```

```
p_atomic:
<   receive action;
    send and local actions;
>
```

Informal Rule

- "A receive action, followed by any number of send and/or internal actions, can be treated as an atomic block of code."

```
p:
    receive action;
    send and local actions;
```

```
p_atomic:
<   receive action;
    send and local actions;
>
```

- *Any property of p_{atomic} is a property of p.*
- Intuition: other actions not visible.

Utility of Informal Rule

- Given a general program, reason about it as if the blocks are atomic.
- If the atomic program is correct,
  the general program is correct too!
- Atomic program easier to reason about (fewer interleavings).

Research Question

- When is this informal rule sound?
  - p_{atomic} is correct ==> p is correct
- What are the requirements on the synchronization system to permit such blocks to be implicitly considered atomic?
  - *i.e.*, what kinds of "receive actions" and "send actions" are needed for this rule to be sound?
  - *e.g.*, in program "Simple", messages are delivered in order

Example: Program $\text{Simple}_{\text{atomic}}$

$\text{p1}_{\text{atomic}}$:
```
<
    ? x
    int a = x + 1
    ! a
    int b = a * a
    ! b
>
```

$\text{p2}_{\text{atomic}}$:
```
<
    int y = 5
    ! y
>
    ? y1
    ? y2
```

Model of Computation

- Many threads of control.
- Each has local storage.
  - not externally visible
  - “local actions” change this storage
- Share a common storage, by which the threads communicate.
  - message-passing layer
  - semaphores

Model of Computation (cont’d)

- Two actions for changing the shared state:
  - an action that *cannot* suspend
    - “send” or “write” action
    - denoted by !
  - an action that *can* suspend (synchronization)
    - “receive” or “read” action
    - denoted by ?
- There are other models.

Related Work: Action Systems

- Lipton and Lamport have considered this problem in the context of action systems.
- Actions map initial states to final states.
- Proved the soundness of a related rule:
  - send actions must commute
  - receive actions must “right commute” with send actions on other processes.

Our Contribution

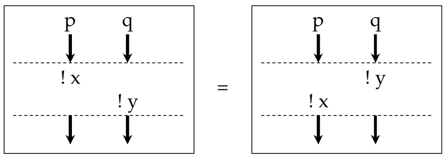
- Action system approach has not considered actions that may or may not terminate.
- Our approach:
  - based on weakest precondition semantics
  - considers actions that may or may not terminate
- Discovery: weaker conditions on send and receive actions.

Computations and Refinement

- A program yields a set of computations.
- Nondeterministic choice within this set.
- A program satisfies a property only if all the possible computations satisfy that property.
- Must establish: subset inclusion.
  - set of general computations is a *subset* of the set of atomic ones
- Follows from three conditions...

Refinement Condition 1

- Sends commute.
  - ➤ the order of two send actions on different processes can be exchanged with no effect.

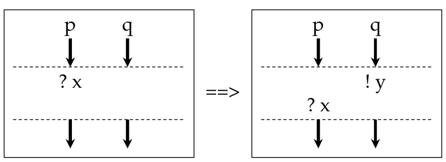


Implications of Condition 1

- Excludes sharing of channels by senders.
  - ➤ broadcast, multicast, shared bus
- Excludes undisciplined modification of shared variables.
  - ➤ semaphore increment actions do commute

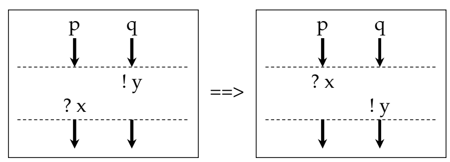
Refinement Condition 2

- Receives are enabled-stable.
  - ➤ if a receive action is enabled, it cannot be disabled by a send on another process



Refinement Condition 3

- Receives are send-monotonic.
  - ➤ a receive action, when swapped with a preceding send action, yields the same (or “stronger”) result



Two Versions of Condition 3

- Strong.
  - ➤ always required
- Weak.
  - ➤ required only when receive is guaranteed to terminate

Refinement Conditions

- Commuting sends
  - ➤ wlp.(p!;q!).Q ==> wlp.(q!;p!).Q
- Enabled-stable receives
  - ➤ wp.p?.true ==> wp.(q!;p?).true
- Weakly send-monotonic receives
  - ➤ wp.p?.true /\ wp.(q!.p?).Q ==> wlp.(p?;q!).Q

- These conditions used to prove that atomic computation is a refinement of general one.
  - they are sufficient, not necessary
- Three examples to consider:
  - probes
  - message passing with bounded buffers
  - shared monotonic counters

Probes: A Dangerous Primitive

- Send action sets a synchronization flag
- Receive action is a probe:
  - when flag is set, returns true
  - otherwise, returns false
- Condition 1: sends commute.
- Condition 2: receives always enabled.
- Condition 3: violated!

Example of Probes

```
p1:

    ! flag1
    ! flag2
```

```
p2:
    repeat
        skip
    until (? flag1)
    ? flag2
```

could be true or false

Example of Probes

```
p1atomic:
<
    ! flag1
    ! flag2
>
```

```
p2atomic:
    repeat
        skip
    until (? flag1)
    ? flag2
```

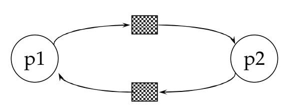
guaranteed to be true

M.P. with Bounded Channels: A Dangerous Primitive

- Channels with finite buffer size.
- Two options for send when buffer is full:
  - suspend (until no longer full)
  - drop the message
- Neither option meets the Refinement Conditions!
  - sends must always be enabled
  - dropping message violates Condition 3

Example of Bounded Channels

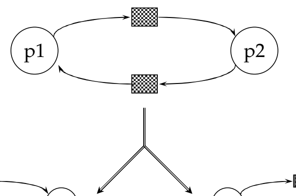
```
p1atomic:
<
    ? x
    ! y
>
```

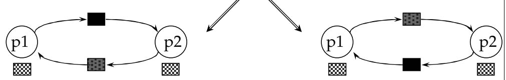



```
p2atomic:
<
    ? y
    ! x
>
```

Example of Bounded Channels

p1$_{atomic}$:
<
? x
! y
>

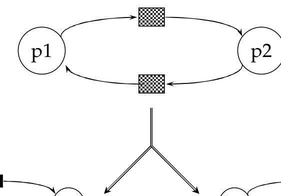


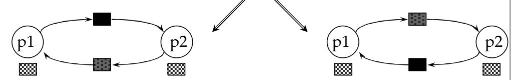


p2$_{atomic}$:
<
? y
! x
>

Example of Bounded Channels

p1:
? x
! y

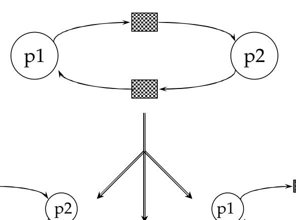


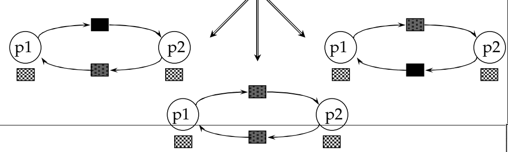


p2:
? y
! x

Example of Bounded Channels

p1:
? x
! y





p2:
? y
! x

Solution for Bounded Channels

- Define send action to be nondeterministic above a certain threshold.
  - *e.g.*, for a buffer size of n, send is nondeterministic when there are n-1 messages
  - nondeterministic send can change the state of the channel arbitrarily
- If the atomic computation does not exceed this threshold, neither does the general one.

Monotonic Counters: A Safe Primitive

- Send action:
  - increase a shared counter by some amount
  - sends commute
- Receive action:
  - suspend until counter reaches some threshold
  - returns a value equal to or less than the current value of the counter.

Example of Monotonic Counters

master :

for all *i*
! $p^i = p^i + 1$

? x >= 5

worker^i :

? $p^i >= 1$
solve subproblem *i*
! x = x + 1

Example of Monotonic Counters

master_{atomic} :
$<$
for all *i*
! $p^i = p^i + 1$
$>$
? $x >= 5$

worker^i_{atomic} :
$<$
? $p^i >= 1$
solve subproblem *i*
! $x = x + 1$
$>$

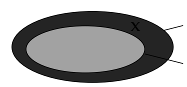
Synchronization System Design

- Synch. system design is often *ad hoc*.
  - familiarity, convenience, efficiency
- *Synchronization primitives must meet the refinement conditions 1 - 3.*
  - conditions are *sufficient* for *safety* properties.
- If including dangerous primitives:
  - distinguish these primitives from safe ones
  - define a discipline making the primitives safe
  - have a good reason

Contact Information

Paul Sivilotti
Dept. of Computer and Information Science
Ohio State University
(614) 292-5835
paolo@cis.ohio-state.edu

Distinguishing Strong and Weak Send-Monotonicity



- Example:
  - receive is nondeterministic when not enabled (may return an arbitrary value or not terminate)
  - weakly send monotonic, but not strongly
- Distinction disappears when recevies are:
  - deterministically terminating
  - nonmiraculous