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.

Outline
- 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.

Atomic Blocks

Multiple Threads of Execution
- Executing
- Waiting
- Atomic Block

Atomic Blocks

- Atomic blocks: 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

<table>
<thead>
<tr>
<th>p1:</th>
<th>p2:</th>
</tr>
</thead>
</table>
| \(? x                   | int y = 5
| int a = x + 1          | ! y
| ! a                    | ? y1
| int b = a * a          | ? y2
| ! b                    |                          |
Example: Program $\text{Simple}_{\text{atomic}}$

1. $p_1_{\text{atomic}}$:
   - $\langle$ x
   - $\text{int } a = x + 1$
   - $\text{!a}$
   - $\text{int } b = a \times a$
   - $\text{!b}$
   - $\rangle$

2. $p_2_{\text{atomic}}$:
   - $\langle$
   - $\text{int } y = 5$
   - $\text{!y}$
   - $\rangle$
   - $\langle$
   - $\text{?y1}$
   - $\text{?y2}$

Informal Rule

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

In informal rule:

- $p: \text{receive action; send and local actions; }$
- $p_{\text{atomic}}: \langle \text{receive action; send and local actions; } \rangle$

Informal Rule

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

In informal rule:

- $p: \text{receive action; send and local actions; }$
- $p_{\text{atomic}}: \langle \text{receive action; send and local actions; } \rangle$

Informal Rule

- “Any property of $p_{\text{atomic}}$ 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?
  - $p_{\text{atomic}}$ is correct $\implies$ $p$ 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 Simple_{atomic}

\[
p_1_{atomic}:
\begin{array}{l}
? x \\
\text{int a = x + 1} \\
! a \\
\text{int b = a * a} \\
! b
\end{array}
\]

\[
p_2_{atomic}:
\begin{array}{l}
\text{int y = 5} \\
! y \\
? y1 \\
? y2
\end{array}
\]

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 \textit{cannot} suspend
    - “send” or “write” action
    - denoted by !
  - an action that \textit{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 \textit{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.

\[
\begin{array}{c}
\text{p} & \text{q} \\
\downarrow & \downarrow \\
?x & !y \\
\downarrow & \downarrow \\
\end{array}
= \quad
\begin{array}{c}
\text{p} & \text{q} \\
\downarrow & \downarrow \\
?x & !y \\
\downarrow & \downarrow \\
\end{array}
\]

### 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

\[
\begin{array}{c}
\text{p} & \text{q} \\
\downarrow & \downarrow \\
?x & !y \\
\downarrow & \downarrow \\
\end{array}
\Rightarrow
\begin{array}{c}
\text{p} & \text{q} \\
\downarrow & \downarrow \\
?x & !y \\
\downarrow & \downarrow \\
\end{array}
\]

### Refinement Condition 3

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

\[
\begin{array}{c}
\text{p} & \text{q} \\
\downarrow & \downarrow \\
?x & !y \\
\downarrow & \downarrow \\
\end{array}
\Rightarrow
\begin{array}{c}
\text{p} & \text{q} \\
\downarrow & \downarrow \\
?x & !y \\
\downarrow & \downarrow \\
\end{array}
\]

### Two Versions of Condition 3

- **Strong.**
  - Always required
- **Weak.**
  - Required only when receive is guaranteed to terminate

### Refinement Conditions

- **Commuting sends**
  - \( wlp.(p!;q!).Q \Rightarrow wlp.(q!;p!).Q \)
- **Enabled-stable receives**
  - \( wp.p?\.true \Rightarrow wp.(q!;p?).true \)
- **Weakly send-monotonic receives**
  - \( wp.p?\.true \wedge wp.(q!;p?).Q \Rightarrow wlp.(p?q!).Q \)
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

guaranteed to be true

Example of Bounded Channels

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

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

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

Example of Bounded Channels

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

```
p2atomic:
<
? y
! x
>```
Example of Bounded Channels

\[ p_1^{\text{atomic}}: \]

\[ ? x \]
\[ ! y \]

\[ p_2^{\text{atomic}}: \]

\[ ? y \]
\[ ! x \]

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.

Example of 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:**

\[ x \geq 5 \]

**Worker[i]:**

\[ ! p_i = p_i + 1 \]

\[ ? x >= 5 \]
### Example of Monotonic Counters

<table>
<thead>
<tr>
<th>master\textsubscript{atomic} :</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
</tr>
<tr>
<td>for all i</td>
</tr>
<tr>
<td>! p = p' + 1</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
<tr>
<td>? x &gt;= 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>worker\textsubscript{atomic} :</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
</tr>
<tr>
<td>? p' &gt;= 1</td>
</tr>
<tr>
<td>solve subproblem i</td>
</tr>
<tr>
<td>! x = x + 1</td>
</tr>
<tr>
<td>&gt;</td>
</tr>
</tbody>
</table>

### 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 receives are:
  - deterministically terminating
  - nonmemorable