Verifying the Buckley-Silberschatz Algorithm for Generalized Input-Output Construct of CSP using the SPIN Model Checker

Vinod Viswanath
Department of Electrical & Computer Engineering
University of Texas at Austin
vvinod@ece.utexas.edu

8 December, 2001
Outline

- Generalized Alternative Commands in CSP
- The Buckley-Silberschatz Algorithm [BS83]
- The SPIN Model Checker
- Verifying Algorithm Properties
Communicating Sequential Processes

\textit{(CSP)} \ [Ho78]

- A CSP program consists of $N$ sequential processes that start simultaneously and terminate when each finishes its last executable statement.
- Each process can access only its own data.
- Communication between processes can happen only when a process’s \textit{output} statement matches the \textit{input} statement of another process.
- Any \textit{input} or \textit{output} statement must name the corresponding process explicitly.
Alternative Command in CSP

- The *alternative command* (based on Dijkstra’s *guarded commands* [Di75]) of CSP allows a process to arbitrarily select one of several statements for processing.

- Every statement is protected by a guard, which is a set of boolean expressions and/or one input statement.

- If guard is enabled, then the statement is considered for execution.

- The *alternative command* allows the execution of a statement if its guard is true.

- If multiple guards are true, then one guard is chosen non-deterministically.
Generalized Alternative Command

- In Hoare’s CSP a process which outputs data *must* commit to that action.

- The *Generalized Alternative Command* allows for output commands in the guard.

- This allows specification of programs with subroutines that contain *call-by-result* parameters.

- Ensures that externally visible effect and behavior of every parallel command can be modeled by some sequential command
Efficiency Criteria for Generalized Commands

- Fewer processes are good
- Lesser information for making communication decisions is good
- Timeout on wait for communication is needed
- Reduced message overhead (by means of efficient algorithm) is good
Buckley-Silberschatz Algorithm

- Process $P_i$ enters an alternative command

- $P_i$ attempts communication with $P_j$ by sending $Query(i, j)$

- While $P_i$ is waiting for response from $P_j$, it cannot
  - initiate $Query(i, k)$ with any other process $P_k$ in the command
  - respond to agree to communicate with any process $P_k$
Buckley-Silberschatz Algorithm (continued . . . )

• In this interval, if $P_i$ receives $Query(k, i)$, then $P_i$
  
  – if ($k > i$) sends $Busy(i, k)$ – a non-committal reply
  
  – if ($k < i$) delays sending a response

• If $P_i$ receives a $Busy(j, i)$ $\Rightarrow$ $P_i$ sent $Query(i, j)$ and ($i > j$) and $P_j$ was waiting for some $Query(j, k)$ to return
Buckley-Silberschatz Algorithm (continued . . .)

- If $P_i$ receives $Busy(j, i)$, then $P_i$ will not communicate with $P_j$ until $P_j$ is done with all its outstanding $Query(j, k)$

- If $P_j$ had sent a $Busy(j, i)$ then it will send $Res(j, i)$ to every process $j$ after all queries $Query(j, k)$ return

- When $P_i$ receives $Res(j, i)$ it will send $Retry(i, j)$ once to resolve the noncommittal answer.
When $P_i$ sends $\text{Retry}(i, j)$, the process $P_j$ is in one of three states:

- $P_j$ might have returned to execution since sending the $\text{Busy}(j, i)$ and will respond with $\text{No}(j, i)$
- $P_j$ might have unsuccessfully tried all its $\text{Query}(j, k)$ and will respond with $\text{Yes}(j, i)$
- $P_j$ might be resolving a $\text{Busy}(k, j)$, in which case it can delay all $\text{Query}(m, j)$ and $\text{Retry}(m, j)$

The sequence of $\text{Retry}$ messages is acyclic

$P_i$ sends only one $\text{Retry}(i, j)$ and is assured of a $\text{Yes}(j, i)$ or a $\text{No}(j, i)$ in response
Algorithm States of $P_i$

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>$P_i$ is executing and not attempting to select a matching communication command</td>
</tr>
<tr>
<td>$Q1$</td>
<td>$P_i$ is in its alternative command and is determining $j$ to send $\text{Query}(i, j)$</td>
</tr>
<tr>
<td>$Q2$</td>
<td>$P_i$ has sent $\text{Query}(i, j)$ and is awaiting an answer</td>
</tr>
<tr>
<td>$R1$</td>
<td>$P_i$ has send all $\text{Query}(i, j)$ and received all answers and is now resolving all $\text{guard}[k] = \text{busy}$</td>
</tr>
<tr>
<td>$R2$</td>
<td>$P_i$ has sent $\text{Retry}(i, j)$ and is awaiting an answer</td>
</tr>
<tr>
<td>$W$</td>
<td>$P_i$ has attempted communication with all $P_j$ and received $\text{No}(j, i)$ from each, and is idling waiting to receive a matching message</td>
</tr>
</tbody>
</table>
Algorithm Data Structures

The $busy_i[j]$ array:

$$busy_i[j] = \begin{cases} 
0 & \text{if } P_i \text{ has not sent } Busy(i, j) \\
1 & \text{if } P_i \text{ has sent } Busy(i, j) \text{ but has not yet sent } Res(i, j) \\
2 & \text{if } P_i \text{ has sent both } Busy(i, j) \text{ and } Res(i, j)
\end{cases}$$
Algorithm Data Structures

The $\text{delay}_i[j]$ array:

$$\text{delay}_i[j] = \begin{cases} 
\text{true} & \text{if } P_i \text{ has received a Query}(j,i) \text{ or a Retry}(j,i) \\
& \text{and } (j < i) \\
& \text{(will not answer until } P_i \text{ receives all of its own responses)}
\end{cases}$$
Algorithm Data Structures

The $\text{guard}_i[j]$ array:

\[
\text{guard}_i[j] = \begin{cases} 
\text{untried} & \text{if } P_i \text{ has not yet sent } \text{Query}(i,j) \\
\text{busy} & \text{if } P_i \text{ has received } \text{Busy}(j,i) \\
\text{no} & \text{if } P_i \text{ has received } \text{No}(j,i) \\
\text{retry} & \text{if } \text{guard}_i[j] \text{ was busy and } P_i \text{ has received } \text{Res}(j,i) \\
\text{yes} & \text{if } P_i \text{ has received } \text{Yes}(j,i) \\
\text{badb} & \text{if } \text{guard}_i[j] \text{ is not enables}
\end{cases}
\]

If $(\text{guard}_i[j] = \text{badb})$ then
the alternative command fails and $P_i$ continues in state $E$. 
### Possible Messages in the Algorithm

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Yes</strong>(i, j):</td>
<td>( P_i ) has an enabled guard naming ( P_j ), and it agrees to establish communication with ( P_j )</td>
</tr>
<tr>
<td><strong>Busy</strong>(i, j):</td>
<td>( P_i ) is in a construct that has a matching statement, but is currently waiting for a response to some ( \text{Query}(i, k) ) and ( j &gt; i )</td>
</tr>
<tr>
<td><strong>No</strong>(i, j):</td>
<td>( P_i ) can continue execution without communicating with ( P_j )</td>
</tr>
</tbody>
</table>
**Possible Messages (continued . . . )**

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Res(i, j)$</td>
<td>$P_i$ sent $Busy(i, j)$ earlier, and $P_j$ can now send a $Retry(j, i)$ and be assured of a definite response</td>
</tr>
<tr>
<td>$Query(i, j)$</td>
<td>$P_i$ has an enabled guard naming $P_j$, and desires communication with $P_j$ and will wait until $P_j$ responds</td>
</tr>
<tr>
<td>$Retry(i, j)$</td>
<td>$P_j$ has given a noncommittal $Busy(j, i)$ in response to $Query(i, j)$ and has since then promised a definite response by $Res(j, i)$ and $P_i$ will try one more time</td>
</tr>
</tbody>
</table>
The *SPIN* Model Checking Tool

- *SPIN* can be used as a full LTL Model Checking System

- *SPIN* works On-the-fly – avoids the construction of a global state graph or a Kripke structure as a prerequisite for any property verification

- Correctness properties to be verified are written directly in LTL

- *SPIN* exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques
Writing Programs in SPIN

- The language of SPIN is PROMELA

- PROMELA is a non-deterministic language, loosely based on Dijkstra’s guarded command language notation

- For input and output constructs PROMELA borrows notation from Hoare’s CSP

- SPIN supports both buffered message passing and shared variables
Example – Dekker’s mutual exclusion program

active proctype p1 ()
{
  do
    try1 = 0;
    do
      :: ( try2 != 0 ) -> break
      :: ( try2 == 0 ) ->
        if
          :: ( turn != 2 ) -> break
          :: ( turn == 2 ) ->
            try1 = 1;
            do
              :: ( turn != 1 ) -> skip
              :: else -> break;
            od;
            try1 = 0;
          fi
      od;
  fi
}
Example – Dekker’s mutual exclusion program

// critical section
cs1 = 1;
skip;
cs1 = 0;
// end critical section
try1 = 1;
turn = 2;
}
Example – Verifying Liveness, Safety

• $\Box (((\text{try} \, 1 = 1) \Rightarrow \Diamond (cs \, 1 = 1)) \& \& ((\text{try} \, 2 = 1) \Rightarrow \Diamond (cs \, 2 = 1)))$

• $\Box \Diamond ((cs \, 1 = 1) \| (cs \, 2 = 1))$

• $\Box \sim ((cs \, 1 = 1) \& \& (cs \, 2 = 1))$

*SPIN* uses never-claim strategy: negates the goal LTL formula.
Verifying Buckley-Silberschatz Algorithm

- If $P_i$ receives a $\text{Query}(j, i)$, then it sends $\{\text{Yes}(i, j), \text{No}(i, j), \text{Busy}(i, j)\}$

- $\Box(\text{Query}(j, i) \Rightarrow \Diamond(\text{Yes}(i, j) \parallel \text{No}(i, j) \parallel \text{Busy}(i, j)))$

- $P_i$ will send out a $\text{Res}(i, j)$ for every $\text{Busy}(i, j)$ it has sent

- $\Box(\text{Busy}(i, j) \Rightarrow \Diamond\text{Res}(i, j))$

- If $P_i$ receives a $\text{Retry}(j, i)$ then it sends $\{\text{Yes}(i, j), \text{No}(i, j)\}$

- $\Box(\text{Retry}(j, i) \Rightarrow \Diamond(\text{Yes}(i, j) \parallel \text{No}(i, j)))$
More properties

- □(Q₁ ⇒ ◊E)
- □(W ⇒ ◊E)
- □(Q₂ ⇒ ◊∼Q₂)
- □(R₂ ⇒ ◊∼R₂)
Partial Conclusions

- The Buckley-Silberschatz protocol is a **hard** algorithm to verify

- **Verified** $\Box(\text{Query}(j,i) \Rightarrow \Diamond(\text{Yes}(i,j) \parallel \text{No}(i,j) \parallel \text{Busy}(i,j)))$

- **Verified** $\Box(\text{Busy}(i,j) \Rightarrow \Diamond\text{Res}(i,j))$

- **Trying to verify** $\Box(\text{Retry}(j,i) \Rightarrow \Diamond(\text{Yes}(i,j) \parallel \text{No}(i,j)))$

- Still trying to verify more properties
References

