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

12 December, 2001

Abstract

Writing distributed algorithms in Hoare’s CSP becomes more convenient if output statements are allowed in the guards of the alternative and iterative commands. The alternative commands with output statements allowed in the guards are called generalized alternative commands. There are a series of papers discussing the implementation of output statements in these generalized alternative commands. The Buckley-Silberschatz algorithm [3] discusses an efficient implementation and gives an algorithm. The algorithm is sufficiently complicated (Figure 3) to warrant a confidence check. In this paper we describe the algorithm, and experiences in trying to verify it using the SPIN model checker. We also prove the correctness of the algorithm for a very simple two process scenario.

1 Introduction

In 1978, C.A.R. Hoare introduced a now-classic paradigm for parallel programming called communicating sequential processes, or CSP [5]. CSP presented a new approach to concurrent programming, and its legacy has continued to influence well founded proposals for concurrent languages, particularly those relying on synchronous message-passing. In CSP, processes are allowed to communicate with one another via synchronous send and receive operations. When one process attempts to send data to some other process, it blocks until there is a simultaneous matching receive by that process (and vice versa). In each case, the send and receive operations explicitly name their destination or source process.

Hoare also provided CSP with an alternative construct, a generalization of the familiar if-then-else. At an alternative a process nondeterministically chooses between several different actions. However, an action can only be chosen if its associated guard, a boolean expression, is true [9]. A key feature of CSP is that the guards for the different branches of an alternative can also be receive operations. A receive operation is true if another process is attempting a matching send. This feature allows a process to wait for input from several other processes simultaneously. The process can take different actions depending on which receive operation is chosen, and perhaps later loop back into the same alternative construct.
Hoare noted the desirability of also allowing send commands as guards as an important feature when he first proposed CSP, because it is useful for describing some applications (e.g., bounded buffers and interleaved conversations [2]) and removes the asymmetry between sends and receives. While the generalization is not known to be required for any applications, it greatly simplifies programming. The construct is in fact as general as needed, since it can easily be used to implement such high-level communication operations as Reppy’s higher-order concurrency [11].

Unfortunately, it is difficult to support send operations in a non-deterministic choice construct. Silberschatz [12] and Van de Snepscheut [13] examined cases where the construct is easy to implement if only certain processes use it. Buckley and Silberschatz [3] presented an overview of several early attempts, as well as detailing their own algorithm. They also made a notable contribution in outlining four criteria for evaluating different implementation approaches:

1. The number of processes involved in a single synchronization event (between matching send and receive commands) should be small.
2. In order to synchronize, a process should not be required to have too much information about the system and environment.
3. If two processes in the system have matching input and output commands, and they do not synchronize with any other processes, then they should eventually synchronize with one another.
4. The number of messages required to establish communication should be small.

The first two criteria stress locality, while the third ensures progress and the last concerns efficiency. Inefficient implementations of the general construct include those that use global information (e.g., a central coordinator) [7], require an unbounded amount of time [4], or use an unbounded amount of communication [1],[8]. Section 2 details the Buckley-Silberschatz algorithm. An overview of the SPIN model checker is given in Section 3. In Section 4 we explain the verification of the Buckley-Silberschatz algorithm in SPIN. Finally some conclusions and discussion is presented in Section 5.

2 The Buckley-Silberschatz Algorithm

The Buckley-Silberschatz algorithm operates on two major behavioral principles. The first principle can be explained thus:

- 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 $P_k$
E: \textit{P}_i \text{ is executing and not attempting to select a matching communication command}

Q1: \textit{P}_i \text{ is in its alternative command and is determining } j \text{ to send } \text{Query}(i, j)

Q2: \textit{P}_i \text{ has sent } \text{Query}(i, j) \text{ and is awaiting an answer}

R1: \textit{P}_i \text{ has sent all } \text{Query}(i, j) \text{ and received all answers and is now resolving all } \text{guard}[k] = \text{busy}

R2: \textit{P}_i \text{ has sent } \text{Retry}(i, j) \text{ and is awaiting an answer}

W: \textit{P}_i \text{ has attempted communication with all } \textit{P}_j \text{ and received } \text{No}(j, i) \text{ from each, and is idling waiting to receive a matching message}

\begin{verbatim}
\begin{align*}
\text{Figure 1: Major states of the Buckley-Silberschatz Algorithm} \\
\item In this interval, if \textit{P}_i \text{ receives } \text{Query}(k, i), \text{ then } \textit{P}_i \\
& \quad \text{if (} k > i \text{) sends } \text{Busy}(i, k) \text{ – a non-committal reply} \\
& \quad \text{if (} k < i \text{) delays sending a response until a definite answer can be sent} \\
\item If \textit{P}_i \text{ receives a } \text{Busy}(j, i) \Rightarrow \textit{P}_i \text{ sent } \text{Query}(i, j) \text{ and } (i > j) \text{ and } \textit{P}_j \text{ was waiting for some } \text{Query}(j, k) \text{ to return} \\
\end{align*}
\end{verbatim}

The second principle is that:

\begin{verbatim}
\begin{align*}
\item If \textit{P}_i \text{ receives } \text{Busy}(j, i), \text{ then } \textit{P}_i \text{ will not communicate with } \textit{P}_j \text{ until } \textit{P}_j \text{ is done with all its outstanding } \text{Query}(j, k) \\
\item If \textit{P}_j \text{ had sent a } \text{Busy}(j, i) \text{ then it will send } \text{Res}(j, i) \text{ to every process } j \text{ after all queries } \text{Query}(j, k) \text{ return} \\
\item When \textit{P}_i \text{ receives } \text{Res}(j, i) \text{ it will send } \text{Retry}(i, j) \text{ once to resolve the noncommittal answer} \\
\item When \textit{P}_i \text{ sends } \text{Retry}(i, j), \text{ the process } \textit{P}_j \text{ is in one of three states:} \\
& \quad \text{P}_j \text{ might have returned to execution since sending the } \text{Busy}(j, i) \text{ and will respond with } \text{No}(j, i) \\
& \quad \text{P}_j \text{ might have unsuccessfully tried all its } \text{Query}(j, k) \text{ and will respond with } \text{Yes}(j, i) \\
& \quad \text{P}_j \text{ might be resolving a } \text{Busy}(k, j), \text{ in which case it can delay all } \text{Query}(m, j) \text{ and } \text{Retry}(m, j) \\
\item The sequence of } \text{Retry} \text{ messages is acyclic} \\
\item \textit{P}_i \text{ sends only one } \text{Retry}(i, j) \text{ and is assured of a } \text{Yes}(j, i) \text{ or a } \text{No}(j, i) \text{ in response} \\
\end{align*}
\end{verbatim}

We now formally present the algorithm by giving the state transitions and message generation functions. The states of the algorithm are grouped together into six major states, which are further differentiated by the values in one of the three arrays. We do this to abbreviate the presentation of the algorithm, because some states have identical state transitions and message generation. The major states of the algorithm are as depicted in Figure 1.
The data structures for $P_i$ are three arrays, initialized to zero, false, and untried when $P_i$ begins to evaluate a generalized alternative command. These arrays are used only when communication is being established. The three arrays are:

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

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

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

Note that, if all $gard_i[j] = badb$, the alternative command fails with no ill effect and $P_i$ continues in state $E$ [5].

All messages are control messages and contain no information. They affect the algorithm only by the message type. The $Query(i,j)$ and $Retry(i,j)$ messages contain the variable type and whether the operation is an input or an output. If these match the command and variable type of an enabled guard of some receiving process $P_j$, then the message is considered valid. All invalid $Query(i,j)$ and $Retry(i,j)$ cannot be used in the alternative command being evaluated, and hence elicit a $No(j,i)$ response. The ensures that the only matches allowed are the ones Hoare specified as correct. The six types of messages and the effect they have on the algorithm is detailed in Figure 2.

The algorithm is presented in Figure 3 [3].

The claim of the Buckley-Silberschatz algorithm is it guarantees that any scheduler that repeatedly tries to get a process to return to execution will eventually succeed. This is because, a process which has entered the $W$ state will respond agreeing to communicate with the first matching $Query(j,i)$ or valid $Retry(i,j)$. This is obviously a desirable characteristic in the algorithm. The algorithm also claims that it is capable of accepting a message in any state. This means that, if all processes attempt to send a $Query(i,j)$ at once, they cannot be blocked.
The chief claim of the Buckley-Silberschatz algorithm is that it is an efficient implementation measuring up to each of the four (aforementioned) efficient implementation criteria. Synchronization of two processes with matching commands is done by the two processes which will communicate. The amount of system information a process must keep is the state of each process names in the alternative or repetitive command, and this information must be kept only for the duration of selecting a communication. Any process takes a finite time to return to execution or enter the wait state, and no two processes with matching communication commands can both be in the wait state. This ensures that two processes will communicate within finite time. Finally, the number of messages one process sends is at $2M$, where $M$ is the number of processes names in the enabled guards of the alternative command. These $2M$ messages generate at most $M + Q$ Res and Retry messages, and hence one process loads the system with at most $3M + Q$ messages, where $Q$ is the number of processes with priority number lesser than $P_i$. The minimum number of messages a process sends is zero, if some process sends it a matching Query as soon as it enters the $Q1$ state, thereby holding with fourth criterion.

The third and the fourth criterion are satisfied by the theorem: Process $P_i$ sends out at most $2M$ unsolicited messages for every execution of an alternative command. The algorithm relies on the fact it proves that a process progresses to either the $E$ execute state or the $W$ wait state within a finite amount of time. Essentially, this makes proofs of deadlock and livelock absence superfluous. The next section gives an overview of the SPIN Model Checking tool, and Section 4 details the verification of the Buckley-Silberschatz algorithm in SPIN.
Table I. State Transitions and Message Generation Functions

<table>
<thead>
<tr>
<th>S_i</th>
<th>Message received</th>
<th>S_{i+1}</th>
<th>Changed data</th>
<th>Messages sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>∅</td>
<td>Q2</td>
<td>∅</td>
<td>QUERY(i, j)</td>
</tr>
<tr>
<td>R1</td>
<td>∅</td>
<td>R2</td>
<td>∅</td>
<td>RETRY(i, j)</td>
</tr>
<tr>
<td>E</td>
<td>QUERY(k, i)</td>
<td>E</td>
<td>∅</td>
<td>NO(i, k)</td>
</tr>
<tr>
<td>Q1</td>
<td>QUERY(k, i)</td>
<td>E</td>
<td>guard[k] := yes, SB2</td>
<td>YES(i, k), MRES</td>
</tr>
<tr>
<td>Q2</td>
<td>QUERY(k, i), k &gt; i</td>
<td>Q2</td>
<td>busy[k] := 1</td>
<td>BUSY(i, k)</td>
</tr>
<tr>
<td>Q2</td>
<td>QUERY(k, i), k &lt; i</td>
<td>Q2</td>
<td>delay[k] := T</td>
<td>∅</td>
</tr>
<tr>
<td>R1 ∪ W</td>
<td>QUERY(k, i)</td>
<td>E</td>
<td>guard[k] := yes</td>
<td>YES(i, k)</td>
</tr>
<tr>
<td>R2</td>
<td>QUERY(k, i)</td>
<td>R2</td>
<td>delay[k] := T</td>
<td>∅</td>
</tr>
<tr>
<td>E ∨ Q1 ∨ Q2</td>
<td>RETRY(k, i)</td>
<td>S_i</td>
<td>∅</td>
<td>NO(i, k)</td>
</tr>
<tr>
<td>(R1 ∨ R2) ∨ BSY2</td>
<td>RETRY(k, i)</td>
<td>E</td>
<td>guard[k] := yes</td>
<td>YES(i, k)</td>
</tr>
<tr>
<td>(R1 ∨ R2) ∨ (W ∨ BSY2)</td>
<td>RETRY(k, i)</td>
<td>S_i</td>
<td>∅</td>
<td>NO(i, k)</td>
</tr>
<tr>
<td>R3 ∨ BSY2</td>
<td>RETRY(k, i)</td>
<td>R2</td>
<td>delay[k] := T</td>
<td>∅</td>
</tr>
<tr>
<td>E ∨ W</td>
<td>RES(k, i)</td>
<td>S_i</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>(Q1 ∨ Q2 ∨ R1 ∨ R2) ∨ BRET</td>
<td>RES(k, i)</td>
<td>S_i</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>(Q1 ∨ Q2 ∨ R1 ∨ R2) ∨ BRET</td>
<td>RES(k, i)</td>
<td>S_i</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>Q3</td>
<td>YES(j, i)</td>
<td>E</td>
<td>guard[j] := yes, SB2</td>
<td>MRES, MD1</td>
</tr>
<tr>
<td>R3</td>
<td>YES(j, i)</td>
<td>E</td>
<td>guard[j] := yes</td>
<td>MD1</td>
</tr>
<tr>
<td>Q2 ∨ BDEL</td>
<td>BUSY(j, i)</td>
<td>E</td>
<td>guard[k] := yes, SB2</td>
<td>MRES, MD2, YES(i, k)</td>
</tr>
<tr>
<td>Q2 ∨ ¬BDEL ∨ BUN</td>
<td>BUSY(j, i)</td>
<td>Q1</td>
<td>guard[j] := busy</td>
<td>∅</td>
</tr>
<tr>
<td>Q2 ∨ ¬BDEL ∨ BUN</td>
<td>BUSY(j, i)</td>
<td>R1</td>
<td>guard[j] := busy, MRES</td>
<td>SB2</td>
</tr>
<tr>
<td>Q1 ∨ BDEL</td>
<td>NO(j, i)</td>
<td>E</td>
<td>guard[k] := yes, SB2</td>
<td>MRES, MD2, YES(i, k)</td>
</tr>
<tr>
<td>Q2 ∨ ¬BDEL ∨ BUN</td>
<td>NO(j, i)</td>
<td>Q1</td>
<td>guard[k] := no</td>
<td>∅</td>
</tr>
<tr>
<td>Q2 ∨ ¬BDEL ∨ BNO</td>
<td>NO(j, i)</td>
<td>W</td>
<td>guard[k] := no, SB2</td>
<td>MRES</td>
</tr>
<tr>
<td>Q2 ∨ ¬BDEL ∨ BYE</td>
<td>NO(j, i)</td>
<td>R1</td>
<td>guard[k] := no, SB2</td>
<td>MRES</td>
</tr>
<tr>
<td>R3 ∨ BDEL</td>
<td>NO(i, j)</td>
<td>E</td>
<td>guard[k] := yes, MD2, YES(i, k)</td>
<td></td>
</tr>
<tr>
<td>R3 ∨ ¬BDEL ∨ BYE</td>
<td>NO(i, j)</td>
<td>R1</td>
<td>guard[k] := no</td>
<td>∅</td>
</tr>
<tr>
<td>R1 ∨ ¬BDEL ∨ BNO</td>
<td>NO(i, j)</td>
<td>W</td>
<td>guard[k] := no</td>
<td>∅</td>
</tr>
</tbody>
</table>

Booleans:                               Messages:
BDEL  ∃k delay[k] = T, k ≠ j       MRES  RES(i, k) ∀b ∃busy[k] = 1
BUN  ∃k guard[k] = untried, k ≠ j  MD1  NO(i, k) ∀b ∃delay[k] = T
BNO  ∀b guard[k] = no ∨ guard[k] = bad, MD2  NO(i, m) ∀m ∃delay[m] = T,
      k ≠ j                                           k ≠ m
BSY2  busy[k] = 2
BRET  guard[k] = busy
BYE   ∃k guard[k] = busy ∨ guard[k] = retry, k ≠ j

Notes:
1. S_i ∨ S_i = choose either S_i for this entry.
2. (S_i ∨ S_i) ∨ B_i = choose either S_i and restrict it to states where B_i is true.
3. SB2 = ∀b if busy[k] = 1 then set busy[k] = 2.
4. All QUERY(k, i) and RETRY(k, i) that do not match a guard in the alternative command are answered NO(i, k). These are not listed in the table.
5. If S_i ∨ BDEL returns to execution, YES(i, k) is sent to P_i ∃delay[k] = T.


Figure 3: The Buckley-Silberschatz Algorithm
3 The SPIN Model Checking Tool

SPIN [6] is a generic verification system that supports the design and verification of asynchronous process systems. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. Process interactions can be specified in SPIN with rendezvous primitives, with asynchronous message passing through buffered channels, through access to shared variables, or with any combination of these three.

As a formal methods tool, SPIN aims to provide:

\(\alpha\) an intuitive, program-like notation for specifying design choices unambiguously, without implementation details,

\(\beta\) a powerful, concise notation for expressing general correctness requirements, and

\(\gamma\) a methodology for establishing the logical consistency of the design choices from \(\alpha\) and the matching correctness requirements from \(\beta\).

In SPIN the notations are chosen in such a way that the logical consistency of a design can be demonstrated mechanically by the tool. SPIN accepts design specifications written in the verification language Promela (a Process Meta Language), and it accepts correctness claims specified in the syntax of standard Linear Temporal Language (LTL) [10]. Promela is a non-deterministic language, loosely based on Dijkstra’s guarded command language notation. It borrows its input and output constructs notation from Hoare’s CSP.

SPIN verification procedure is based on an optimized depth-first graph traversal method. The cycle detection mechanism is based on the classical Tarjan’s depth-first search algorithm, and is called nested depth-first search. The nested depth-first search does not detect all possible acceptance cycles that may appear in the reachability graph. It can, however, be proved to detect at least one such cycle if it exists. Since acceptance cycles in SPIN are counter examples to the correctness claims (never-claim strategy), establishing either the absence or presence of cycles is sufficient for purposes of verification.

SPIN used a partial order reduction method to reduce the number of reachable states that need to be explored to complete a verification. The reduction is based on the general observation that the validity of an LTL formula is often insensitive to the order in which concurrent and independently executed events are interleaved in the depth-first search. Therefore, instead of generating an exhaustive state space that includes all execution sequences as paths, the verifier can generate a reduced state space, with only representatives of classes of execution sequences that are indistinguishable for a given correctness property.

SPIN also uses bit-state hashing and state-compression techniques to economize on memory requirements of a reachability analysis, without incurring unrealistic increases in runtime requirements. The next section describes representing the Buckley-Silberschatz algorithm (Figure 3) in Promela and proving theorems about its correctness using SPIN.
Verifying Buckley-Silberschatz Algorithm in \textit{SPIN}

In order to verify the Buckley-Silberschatz Algorithm in \textit{SPIN}, we came up with many properties of the Buckley-Silberschatz which need to hold to guarantee the correctness of the algorithm.

\textbf{Property 1} If \(P_i\) receives a \textit{Retry}(\(j, i\)) then it sends a \{\textit{Yes}(\(i, j\)), \textit{No}(\(i, j\))\} within a finite time.

Buckley and Silberschatz give a proof argument for this property \cite{3} based on the state of \(P_i\). Process \(P_i\), when it receives the \textit{Retry}(\(j, i\)), can be in one of five states, and all the scenarios can be addressed in three cases:

- If \(P_i\) is in one of \{\textit{E}, \textit{Q1}, \textit{Q2}\} states and \textit{Retry}(\(j, i\)) is true, then \(P_i\) sends a \textit{No}(\(i, j\)).
- If \(P_i\) is in one of \{\textit{R1}, \textit{W}\} states and \textit{Retry}(\(j, i\)) is true, then \(P_i\) will send one of \{\textit{Yes}(\(i, j\)), \textit{No}(\(i, j\))\}.
- If \(P_i\) is in state \textit{R2} and \textit{Retry}(\(j, i\)) is true, then \(P_i\) will respond within finite time, and the answer is dependent on the priority of \(P_i\) with respect to \(P_j\).

The LTL formula representing Property 1 can be written as:

\textbf{LTL Proposition 1} \(\Box(\text{\textit{Retry}(\(j, i\))} \Rightarrow \Diamond(\text{\textit{Yes}(\(i, j\)) || \textit{No}(\(i, j\)))})\)

\textbf{Property 2} If \(P_i\) receives a \textit{Query}(\(j, i\)), then it sends a \{\textit{Yes}(\(i, j\)), \textit{No}(\(i, j\)), \textit{Busy}(\(i, j\))\} within a finite time.

Again, Buckley and Silberschatz give a proof argument for this \cite{3} proposition considering four cases,

- If \(P_i\) is in state \textit{E} and receives a \textit{Query}(\(j, i\)), then it sends a \textit{NO}(\(i, j\)).
- If \(P_i\) is in one of \{\textit{R1}, \textit{Q1}, \textit{W}\} states and receives a \textit{Query}(\(j, i\)), then it sends either \textit{Yes}(\(i, j\)) or \textit{No}(\(i, j\)) within finite time.
- If \(P_i\) is in \textit{R2} waiting for \(P_k\), then by Property 1 \(P_i\) will receive either \textit{Yes}(\(i, j\)) or \textit{No}(\(i, j\)) within finite time from \(P_k\), and will correspondingly respond \textit{No}(\(i, j\)) or \textit{Yes}(\(i, j\)) within finite time.
- If \(P_i\) is in state \textit{Q2} and receives a \textit{Query}(\(j, i\)), then \(P_i\) will respond within finite time, and the answer is dependent on the priority of \(P_i\) with respect to \(P_j\).

The LTL formula representing Property 2 can be written as:

\textbf{LTL Proposition 2} \(\Box(\text{\textit{Query}(\(j, i\))} \Rightarrow \Diamond(\text{\textit{Yes}(\(i, j\)) || \textit{No}(\(i, j\)) || \textit{Busy}(\(i, j\)))})\)

\textbf{Property 3} \(P_i\) will send out a \textit{Res}(\(i, j\)) for every \textit{Busy}(\(i, j\)) it has sent within a finite time.

\(P_i\) will send a \textit{Res}(\(i, j\)) to all processes on its \textit{busy} list by one of these transitions \{\textit{Q1} \rightarrow \textit{E}, \textit{Q2} \rightarrow \textit{E}, \textit{Q2} \rightarrow \textit{R1}\}. From the previous property (Property 2) the queries sent out are answered within finite time, and therefore one of these transitions occur within finite time.
LTL Proposition 3 \( \Box(Busy(i,j) \Rightarrow \Diamond Res(i,j)) \)

These properties are essentially based on the theorem mentioned in Section 2. More formally,

Theorem 1 Process \( P_i \) sends out at most \( 2M \) unsolicited messages (\( M \) is the number of processes names in \( P_i \)'s alternative command) per every execution of an alternative command.

Some other properties that we tried verifying were:

Property 4 If a process \( P_i \) enters its alternative command then eventually it will start executing one of its named instructions.

The LTL formula for this proposition is given by \( \Box(Q1 \Rightarrow \Diamond E) \)

Property 5 If a process \( P_i \) enters a state \( W \), then eventually (within finite time) it will start executing one of its named instructions.

The LTL formula for this proposition is given by \( \Box(W \Rightarrow \Diamond E) \)

Property 6 If a process \( P_i \) enters a state \( Q2 \) (\( P_i \) has sent all its \( Query(i,j) \) and is waiting for responses), then within finite time it will receive all its responses and move out of state \( Q2 \).

The LTL formula for this proposition is given by \( \Box(Q2 \Rightarrow \Diamond \sim Q2) \)

Property 7 If a process \( P_i \) enters a state \( R2 \) (\( P_i \) has sent all its \( Retry(i,j) \) and is waiting for responses), then within finite time it will receive all its \( Yes(j,i) \) or \( No(j,i) \) responses and move out of state \( R2 \).

The LTL formula for this proposition is given by \( \Box(R2 \Rightarrow \Diamond \sim R2) \)

Another theorem (failure of which implies deadlock) that we tried verifying was:

Theorem 2 If \( P_i \) enters the \( W \) state, then there is no other \( P_i \) with a matching enabled guard for \( P_i \), that is also in the state \( W \).

Results of verifying all these properties and theorems in \( SPIN \) are tabulated in Figure 4.

All verification in \( SPIN \) were tried out for two processes. Initially we tried to run for 10 processes, but the number of LTL formulae that need to be verified in \( SPIN \) grow exponentially with the number of processes. Finally the Buckley-Silberschatz algorithm was verified for the elementary two-process base case scenario.
| Property 1    | □(Retry(j, i) ⇒ ◇(Yes(i, j) || No(i, j))) | Verified |
|--------------|------------------------------------------|----------|
| Property 2   | □(Query(j, i) ⇒ ◇(Yes(i, j) || No(i, j) || Busy(i, j))) | Verified |
| Property 3   | □(Busy(i, j) ⇒ ◇Res(i, j)) | Verified |
| Property 4   | □(Q1 ⇒ ◇E) | Verified |
| Property 5   | □(W ⇒ ◇E) | Verified |
| Property 6   | □(Q2 ⇒ ◇Q2) | Verified |
| Property 7   | □(R2 ⇒ ◇R2) | Verified |

Figure 4: Results of verifying properties of Buckley-Silberschatz algorithm in SPIN.

5 Conclusions and Discussion

From the perspective of a verification engineer, the Buckley-Silberschatz algorithm is a very complicated algorithm. It is not clear that the properties verified and tabulated in Figure 4 are complete, i.e., they guarantee the correctness of the algorithm over the entire possible state space. Also it is not very clear if an algorithm which is correct for 2-processes, is guaranteed to be correct for > 2-processes. Infact, history suggests that requirements for correctness of 4-process scenario is strictly greater than for the 3-process scenario which in turn is strictly greater than for the 2-process scenario. We conjecture that there is a finite bound n, such that, if we prove the correctness for all scenarios <= n-processes, then the algorithm will be correct independent on the number of processes.

References