Automatic Refinement of Split Binary Semaphore (extended version)

Damián Barsotti and Javier O. Blanco
Fa.M.A.F., Universidad Nacional de Córdoba,
Córdoba 5000, Argentina

Abstract. Binary semaphores can be used to implement conditional critical regions by using the split binary semaphore (SBS) technique. Given a specification of a conditional critical regions problem, the SBS technique provides not only the resulting programs but also some invariants which ensure the correctness of the solution. The programs obtained in this way are generally not efficient. However, they can be optimized by strengthening these invariants and using them to eliminate unnecessary tests.

We present a mechanical method to perform these optimizations. The idea is to use the backward propagation technique over a guarded transition system that models the behavior of the programs generated by the SBS. This process needs proving heavy implications and simplifying growing invariants. Our method automatically entrusts these tasks to the Isabelle theorem prover and the CVC Lite validity checker. We have tested our method on a number of classical examples from concurrent programming.

1 Introduction

Split Binary Semaphores (SBS) are used to implement conditional critical regions [4, 11]. Since the method is fairly general, the solutions obtained often perform some unnecessary tests when leaving a critical region. This work focuses on the mechanical elimination of these unnecessary tests in the programs.

The SBS technique provides not only the programs implementing conditional critical regions, but also some invariants which ensure its correctness. Starting from them, we find stronger invariants from which it can be automatically deduced that certain tests will always be false and hence can be eliminated.

In order to achieve these optimizations, we model programs as guarded transition systems, for which many invariant generation techniques are well-known. In this work we mainly use the backward propagation technique [3] which provides invariants that are quantifier-free formulae. The proofs involved in their manipulation can be dealt with fully automated provers\(^1\) such as CVC Lite [2].

\(^1\) Some provers (like CVC Lite and Simplify) include some support for first-order quantifiers elimination. Nevertheless, these provers work better and faster on quantifier-free formulae since support for quantifiers is not complete.
This validity checker also supports several interpreted theories (including rational and integer linear arithmetic, arrays, tuples, etc.) which are appropriate for obtaining assertions on programs.

The propagation techniques (backward and forward) are based on the calculus of a fixed point of a formula transformer. One of the advantages of the backward propagation technique is that the sequence of approximations is usually finite. Unfortunately, the formulae produced during this process are large. We use some simplification techniques implemented in CVC Lite and in the Isabelle theorem prover [10] to tackle this problem. Isabelle is an interactive theorem prover that offers efficient tactics for simplification. Besides, we implement other simplification methods using CVC Lite.

The paper is structured as follows. In Sec. 2 we explain the SBS technique. In Sec. 3 we present an example on the use of this technique. Sec. 4 and 5 present the theoretical framework for constructing the guarded transition system and the method for obtaining the invariants. The guarded transition system that results from the programs generated by SBS technique is developed in Sec. 6 and the refinement procedure in Sec. 7. In Sec. 8 we show some optimizations applied on the refinement procedure. Finally, we show some examples in Sec. 9 and expose our conclusions and suggestions for further work in Sec. 10.

2 Split Binary Semaphores

It is well-known that binary semaphores can easily ensure mutual exclusion and are suitable to implement critical regions. Moreover, they can be used in a systematic way to implement conditional critical regions. We briefly present the solution and refer to the literature for a justification of the method [4, 1, 11, 9].

A set \( \{s_0, \ldots, s_n\} \) of binary semaphores is called a split binary semaphore if at any time at most one of them equals 1, i.e. the following property is a global invariant for the (multi)program:

\[
0 \leq \langle \sum_i : 0 \leq i \leq n : s_i \rangle \leq 1
\]

In the execution of the program, every critical region begins with a P on some semaphore, and ends with a V operation on some semaphore (not necessarily the same one). Hence, the invariant ensures mutual exclusion between any pair P-V.

Besides ensuring mutual exclusion, SBS satisfies the following domino rule [9]. In an execution of the program, if the last V operation is over semaphore \( s \), then the next P operation will be on the same semaphore (notice that there may be more than one P on \( s \)). This means that the precondition of a V operation can be taken as the postcondition of any corresponding P operation since shared variables can only be modified inside critical sections. This rule can be formulated as the global invariant \( \varphi_{SBS} : \langle \forall s :: s = 0 \lor I_s \rangle \) where \( I_s \) is the assertion that holds before every V.s statement and after every corresponding P.s statement.

In order to implement conditional critical regions, the SBS can be used in the following way. To every different condition, a binary semaphore of the SBS
is associated, and another “neutral” semaphore is used for the case in which no
condition holds. Then, every critical region will be prefixed by a P operation on
the semaphore associated with its precondition. Some care should be taken to
introduce V operations to ensure progress. We illustrate the method with two
conditional critical regions.

Suppose we want to execute statements $S_0$, $S_1$ atomically under conditions
$B_0, B_1$ respectively. Furthermore, the critical regions must preserve a global in-
vARIANT $I$. We use binary semaphores $s_0, s_1$ for each condition and two counters
$b_0, b_1$ which will be used to count the number of processes committed to execute
$P_{s_0}, P_{s_1}$ respectively. These counters are necessary to avoid deadlocks. Another
semaphore $m$ is used for the case in which no condition holds or there is no
process waiting.

The following invariant characterizes the SBS solution for the critical region
problem.

$$
\varphi_{SBS} : (s_0 = 0 \lor (B_0 \land 0 < b_0 \land I)) \land
(s_1 = 0 \lor (B_1 \land 0 < b_1 \land I)) \land
(m = 0 \lor ((\neg B_0 \lor 0 = b_0) \land (\neg B_1 \lor 0 = b_1) \land I)) .
$$

Fig. 1 shows the (fully annotated) component obtained by the application of
the SBS technique for the atomic execution of $S_0$.

This solution can be simplified by distributing both assignments to $b_0$ (the
increment can be distributed as well since $b_0$ does not appear in $B_0$), and then
simplify the first alternative (the P-V pair and afterwards the complementary assignments). The simplified solution is in Fig. 2.

\[
\text{SCC}_0
\]

\[
P.m;
\{ I \land (\neg B_0 \lor b_0 = 0) \land (\neg B_1 \lor b_1 = 0) \}
\]

\[
\text{if } B_0 \rightarrow \{ I \land B_0 \land b_0 = 0 \land (\neg B_1 \lor b_1 = 0) \}
\]

\[
\text{skip}
\]

\[
\Box \neg B_0 \rightarrow \{ I \land \neg B_0 \land (\neg B_1 \lor b_1 = 0) \}
\]

\[
b_0 := b_0 + 1 ;
\]

\[
\{ I \land (\neg B_0 \lor b_0 = 0) \land (\neg B_1 \lor b_1 = 0) \}
\]

\[
V.m ;
\]

\[
P.s_0 ;
\]

\[
\{ I \land B_0 \land b_0 > 0 \}
\]

\[
b_0 := b_0 - 1
\]

\[
\fi
\]

\[
\{ I \land B_0 \}
\]

\[
S.0 ;
\]

\[
\{ I \}
\]

\[
\text{if } B_0 \land b_0 > 0 \rightarrow \{ I \land B_0 \land b_0 > 0 \} \ V.s_0
\]

\[
\Box B_1 \land b_1 > 0 \rightarrow \{ I \land B_1 \land b_1 > 0 \} \ V.s_1
\]

\[
\Box (\neg B_0 \lor b_0 = 0) \land (\neg B_1 \lor b_1 = 0)
\]

\[
\rightarrow \{ I \land (\neg B_0 \lor b_0 = 0) \land (\neg B_1 \lor b_1 = 0) \} \ V.m
\]

\[
\fi
\]

\textbf{Fig. 2.} SBS generated component

3 Bounded Buffer

The following example shows how the simplifications can be manually done. Nevertheless, we apply the simplifications methodically in order to mechanise them later.

Let’s consider the classical \textit{Producer/Consumer} problem through a bounded buffer. The component called \textit{producer} produces some elements and send them to the \textit{consumer} component which will use them. Some synchronisation is needed to avoid writing on a full buffer or reading from an empty one. The formal specification of this problem can be stated as follows.

<table>
<thead>
<tr>
<th>Pre: ( p = 0 \land d = 0 )</th>
<th>Cons: (*[ { ? 0 &lt; p - d } )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prod: *{ { produce.n )</td>
<td></td>
</tr>
<tr>
<td>; { ? p - d &lt; N } )</td>
<td></td>
</tr>
<tr>
<td>; \text{buf.}(p \text{ mod } N) := n )</td>
<td></td>
</tr>
<tr>
<td>; p := p + 1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cons: *{ { m := \text{buf.}(d \text{ mod } N) )</td>
</tr>
<tr>
<td></td>
<td>; d := d + 1 )</td>
</tr>
<tr>
<td></td>
<td>; consume.m )</td>
</tr>
</tbody>
</table>

For a solution with an SBS the access to the buffer will be performed in a critical region. Whereas this is unnecessary for one reader and one writer, it is
required for multiple readers or writers. In order to avoid irrelevant details, we deal only with the synchronisation, leaving out the actual update of the buffer. It can be introduced easily in the final solution. Furthermore, since we are only interested in the difference between \( p \) and \( d \), we transform coordinates to a single integer variable \( n \) which should be invariably equal to \( p - d \). The problem then becomes symmetric.

\[
\begin{array}{|c|c|}
\hline
\text{Pre: } & n = 0 \\
\text{Prod: } & \{ \{ n < N \} \}, \quad n := n + 1 \\
\text{Cons: } & \{ \{ 0 < n \} \}, \quad n := n - 1 \\
\hline
\end{array}
\]

Since the atomic regions are now a single statement, the following global invariant characterises the problem

\[ I : 0 \leq n \leq N . \]

We use a split binary semaphore with three components, one for each condition and one neutral semaphore to ensure mutual exclusion when the conditions do not hold or there are not waiting processes.

Following the general scheme proposed earlier, the semaphore invariants then become:

\[
\varphi_{SBS} : (s = 0 \lor (n < N \land n < b)) \land \\
(t = 0 \lor (0 < n \land 0 < c)) \land \\
(m = 0 \lor ((n = N \lor 0 = b) \land (0 = n \lor 0 = c))) .
\]

Note that the condition are simplified as much as possible using the global invariant.

This solution can be improved by removing some of the guards which we can prove will never be chosen. This simplify the evaluation of the guards when leaving a critical region, and hence it can greatly improve the performance, since in a typical case a critical region will be small but it will be entered many times.
In this example we elaborate from [6] tailoring our proof to a partial mechanisation.

The first simple observation is that after an item is produced the buffer cannot be empty. In order to use this idea, we propose to show the correctness of the assertion with the question mark Prod

\[
\begin{align*}
   & \text{if } n < N \rightarrow \text{skip} \\
   & \Box n = N \rightarrow b := b + 1; \ V.m ; \ P.s ; b := b - 1 \\
   & \mathbf{fi}: \\
   & n := n + 1; \\
   & \{0 < n?\} \\
   & \text{if } n < N \land 0 < b \rightarrow \text{V.s} \\
   & \Box 0 < n \land 0 < c \rightarrow \text{V.t} \\
   & \Box (n = N \vee 0 = b) \land (0 = n \vee 0 = c) \rightarrow \text{V.m} \\
   & \mathbf{fi}
\end{align*}
\]

This is easily proved since \(wp.(n := n + 1)(0 < n)\) equals \(0 \leq n\) which is implied by the global invariant. The guards of the second if can now be simplified. Symmetrically, condition \(n < N\) will be a valid preassertion of the if for the consumer.

Looking operationally at the program, it seems unlikely that a producer may release another producer. This implies that it is possible that the first guard of the final if in the producer can be eliminated. Actually, a more involved argument is needed to fully justify this, but one advantage of the method advocated here is that if we believe that certain property may hold we can try it out and see what we can achieve. The process that we will describe now will be fully automatized (with one minor caveat).

In order to simplify this guard we need the assertion with the question mark to hold:
Prod

\begin{align*}
P.m; \\
&\text{if } n < N \rightarrow \text{skip} \\
&\quad \square n = N \rightarrow b := b + 1; V.m ; P.s ; b := b - 1 \\
&\text{fi} \\
&n := n + 1; \\
&\{0 < n\}\{n = N \land 0 = b?\} \\
&\text{if } n < N \land 0 < b \rightarrow V.s \\
&\quad \square 0 < c \rightarrow V.t \\
&\quad (n = N \lor 0 = b) \land 0 = c \rightarrow V.m \\
&\text{fi} \\
&n := n + 1; \\
&\{0 < n\}\{n < N\} \\
&\text{if } 0 < c \rightarrow V.t \\
&\quad \square 0 = c \rightarrow V.m \\
&\text{fi}
\end{align*}

Taking the \textit{wp} we can see that \(n + 1 = N \lor 0 = b\) should hold after the first \textit{if}. In the first branch it follows from \(b = 0\). In the second branch we need \(n + 1 = N \lor 1 = b\) as a valid postcondition for the \(P.s\). This can be achieved by strengthening the invariant for \(s\) to \(s = 0 \lor (0 < b \land n < N \land (n+1 = N \lor 1 = b))\).

In [6] a slightly stronger and simpler invariant is used. This one, however, is the weakest it can be obtained and one of the obvious candidates for a mechanical calculation. The other possible candidate is the strongest one: we will pursue this idea in another work (see the conclusions).

We have to show now that this new assertion is indeed a valid invariant. The only place where it can be invalidated is before the \(V.s\) in the consumer. Taking \textit{wp} we can see that the invariant for semaphore \(t\) has to be strengthened as well into \(t = 0 \lor (0 < c \land 0 < n \land (b \leq 1 \lor n = N))\). Furthermore, the last guard can be simplified provided that the required condition holds.

The annotated program then becomes

\begin{align*}
\text{Prod} & & \text{Cons} \\
P.m; & & P.m; \\
\text{if } n < N \rightarrow \text{skip} & & \text{if } 0 < n \rightarrow \text{skip} \\
\square n = N \rightarrow b := b + 1; V.m ; P.s ; b := b - 1 & & \quad \square 0 = n \rightarrow c := c + 1; V.m ; P.t ; c := c - 1 \\
\text{fi} & & \text{fi} \\
& & n := n - 1; \\
& & \{n < N\} \\
& & \text{if } 0 < b \rightarrow V.s \\
& & \quad \square 0 < n \land 0 < c \rightarrow \\
& & \quad \{b \leq 1 \lor n = N?\} V.t \\
& & \quad \square 0 = b \land (0 = n \lor 0 = c) \rightarrow V.m \\
& & \text{fi}
\end{align*}

The remaining proof obligation can be ignored since we plan to eliminate this branch of the \textit{if} altogether. This is easily done by ensuring the condition \(0 = n \lor 0 = c\) prior to the last \textit{if} of the consumer. The whole process is symmetrical to the one we have just described. At the end we obtain the following invariant
for the condition semaphores, whereas the invariant for the mutual exclusion \( m \) remains unchanged:

\[
\begin{align*}
    s &= 0 \lor 0 < b \land n < N \land (n + 1 = N \lor 1 = b) \land (c \leq 1 \lor 0 = n) \\
t &= 0 \lor 0 < c \land 0 < n \land (1 = n \lor 1 = c) \land (b \leq 1 \lor n = N)
\end{align*}
\]

whereas the whole program (without annotations) becomes

\[
\text{Prod}
\begin{align*}
    \text{P.m;} \\
    \text{if } n < N \to \text{skip} \\
    \square n = N \to b := b + 1; \text{V.m} ; \text{P.s} ; b := b - 1 \\
    \bar{f}: \\
    n := n + 1; \\
    \text{if } 0 < c \to \text{V.t} \\
    \square 0 = c \to \text{V.m} \\
    \bar{f}
\end{align*}
\]

\[
\text{Cons}
\begin{align*}
    \text{P.m;} \\
    \text{if } 0 < n \to \text{skip} \\
    \square 0 = n \to c := c + 1; \text{V.m} ; \text{P.t} ; c := c - 1 \\
    \bar{f}: \\
    n := n - 1; \\
    \text{if } 0 < b \to \text{V.s} \\
    \square 0 = b \to \text{V.m} \\
    \bar{f}
\end{align*}
\]

The whole process of simplifying these components is carried out (almost) automatically by our method. The user needs only to choose which are the guards that will be eliminated and the system will calculate the new invariants.

4 Guarded Transition System

The following definitions are taken from [12, 3]. Let \( \Sigma \) be a first-order language containing interpreted symbols for concrete domains like booleans, integers and reals. Let \( \mathcal{F} \) be the set of first-order formulas with free (typed) variables contained in a finite set \( \mathcal{V} = \{x_1, \cdots, x_n\} \) over \( \Sigma \). We shall denote the sequence of variables \( x_1, \cdots, x_n \) by \( \bar{x} \).

The usual way to model reactive systems is by using a guarded transition system \( \mathcal{S} = (\mathcal{V}, \Theta, \mathcal{T}) \) where \( \Theta \in \mathcal{F} \) is its initial condition and \( \mathcal{T} \) is a finite set of guarded transitions. Each \( \tau \in \mathcal{T} \) can be specified as follows:

\[
\gamma_{\tau} \implies \bar{x} := \bar{e}_{\tau}(\bar{x})
\]

where \( \gamma_{\tau} \in \mathcal{F} \) is the transition guard and \( \bar{e}_{\tau}(\bar{x}) \) is a sequence of expressions in \( \Sigma \) whose free variables are taken from \( \mathcal{V} \); both sequences should have equal length. The formula \( \gamma_{\tau} \) denotes the condition that should hold in order to execute the transition, \( \bar{x} := \bar{e}_{\tau}(\bar{x}) \) is a simultaneous assignment which indicate the transformation of the state produced by the transition.

Usually, a transition system has a control variable \( vc \in \mathcal{V} \) which ranges over a finite set \( \mathcal{L} \) of locations. This variable acts as a program counter and the locations act as source and target of each transition. Hence, any \( \tau \in \mathcal{T} \) can be written as:

\[
vc = l_{\tau} \land \gamma_{\tau} \implies \bar{x} := \bar{e}_{\tau}(\bar{x}) ; vc := l'_{\tau}
\]
where $l_\tau, l'_\tau \in \mathcal{L}$, $\gamma_\tau$ is a predicate with variables in $\mathcal{V}/\{vc\}$ and $\bar{x} := \bar{e}_\tau(\bar{x})$ is a simultaneous assignment with variables in $\mathcal{V}/\{vc\}$. For a given transition $\tau$ we define the functions $src(\tau) = l_\tau$ and $tgt(\tau) = l'_\tau$.

For each transition, we define a predicate $\Phi_\tau$ as follows:

$$\Phi_\tau(\bar{x}, \bar{x}') \triangleq \gamma_\tau(\bar{x}) \land \bar{x}' = \bar{e}_\tau(\bar{x})$$

where $\bar{x}'$ is the renaming of variables $x_i$ in $\bar{x}$ as $x'_i$. Note that $\Phi_\tau$ is a formula with free variables in $\mathcal{V} \cup \mathcal{V}'$ where $\mathcal{V}'$ is the set of variables obtained by renaming the variables $x$ to $x'$ in $\mathcal{V}$. These predicates denote the relation between the values of the variables before execution of the transitions (in a given location $src(\tau)$), with the values of the same variables after its execution (in location $tgt(\tau)$). Hence, we can define the transition predicate $\Phi$ of a guarded transition system $S$ as follows:

$$\Phi(\bar{x}, vc, \bar{x}', vc') \triangleq \bigvee_{\tau \in \mathcal{T}} vc = src(\tau) \land vc' = tgt(\tau) \land \Phi_\tau(\bar{x}, \bar{x}') .$$

The formula transformer weakest precondition\cite{3,12} of a transition $\tau$, denoted by wp$(\Phi_\tau)$, is defined as follows:

$$wp(\Phi_\tau)(\varphi(\bar{x})) \triangleq \forall \bar{x}', (\Phi_\tau(\bar{x}, \bar{x}') \rightarrow \varphi(\bar{x}')) .$$

The universal quantifier can be eliminated using substitution:

$$wp(\Phi_\tau)(\varphi(\bar{x})) = \gamma_\tau(\bar{x}) \rightarrow \varphi(\bar{e}_\tau(\bar{x})) . \quad (1)$$

In general, the weakest precondition transformer for the whole transition system $S$, denoted by WP$(\Phi)$, is defined as follows:

$$WP(\Phi)(\phi(\bar{x}, vc)) \triangleq \forall \bar{x}', vc'. (\Phi(\bar{x}, vc, \bar{x}', vc') \rightarrow \phi(\bar{x}', vc')) .$$

Since the set of location $\mathcal{L}$ is finite, the predicate $\phi$ can be written as:

$$\phi(\bar{x}, vc) = \bigwedge_{l \in \mathcal{L}} vc = l \rightarrow \phi(\bar{x}, l) . \quad (2)$$

If we define the predicates $\phi_l(\bar{x}) = \phi(\bar{x}, l)$ for each location $l \in \mathcal{L}$ then the weakest precondition of the whole system is equivalent to:

$$WP(\Phi)(\phi(\bar{x}, vc)) = \bigwedge_{\tau \in \mathcal{T}} vc = src(\tau) \rightarrow WP(\Phi_\tau)(\phi_{tgt(\tau)}(\bar{x})) \quad (3)$$

or, using (1):

$$WP(\Phi)(\phi(\bar{x}, vc)) = \bigwedge_{\tau \in \mathcal{T}} (vc = src(\tau) \land \gamma_\tau(\bar{x})) \rightarrow \phi_{tgt(\tau)}(\bar{e}_\tau(\bar{x})) . \quad (4)$$
5 Invariants

Let $\mathcal{S} = \langle \mathcal{V}, \Theta, T \rangle$ be a guarded transition system and $\mathfrak{R}$ a first-order theory over the language $\Sigma$. A formula $\varphi \in \mathcal{F}$ is an inductive invariant if $\mathfrak{R} \models \Theta \rightarrow \varphi$ and $\mathfrak{R} \models \varphi \rightarrow \text{WP}(\Phi)(\varphi)$, with $\Phi$ the transition predicate of the guarded transition system. Since the theory $\mathfrak{R}$ is fixed, we shall not mention it explicitly when we talk about satisfiability and validity in $\mathfrak{R}$. Thus, validity in $\mathfrak{R}$ is denoted by $\models$.

A formula $\phi \in \mathcal{F}$ is an invariant if there exists an inductive invariant $\varphi$ such that $\models \varphi \rightarrow \phi$. This characterization is sound and relative complete.

For a monotone formula transformer $\Gamma : \mathcal{F} \mapsto \mathcal{F}$, such as $\text{WP}$, we write the greatest fixed point as $\nu X. \Gamma(X)$. Its meaning is the usual $\nu$. Given a formula $\phi$, we define the monotone formula transformer $B$ by

$$B(Y) \triangleq \phi \land \text{WP}(\Phi)(Y) \ .$$

(5)

The greatest fixed point $\varphi_B : \nu X. B(X)$ provides the weakest formula $\varphi_B$ satisfying $\models \varphi_B \rightarrow \phi$ and $\models \varphi_B \rightarrow \text{WP}(\Phi)(\varphi_B)$. Therefore, if $\models \Theta \rightarrow \varphi_B$ then $\phi$ is an invariant and $\varphi_B$ is an inductive invariant of the system.

Since $B$ is monotone, if the sequence starting from $\text{True}$

$$\text{True} \leftarrow B(\varphi_0) \leftarrow B(\varphi_1) \leftarrow \cdots$$

converges in finitely many steps, then its limit is $\varphi_B$. From this property we can explore the state space using the backward propagation technique [3]: given a candidate invariant $\phi$, we can find the fixed point $\varphi_B$ if the sequence converges in finitely many steps. Then, proving $\models \Theta \rightarrow \varphi_B$ we can verify if $\phi$ is an invariant of the system.

6 SBS as a Transition System

In this section we explain how to obtain a transition system for a set of processes executing the programs generated by the implementation of conditional critical regions using the SBS technique.

There is a common way to construct transition systems which model the behavior of concurrent programs [8]. The main idea is to first obtain a transition system for every individual thread or sequential component (in a standard way) and then construct their parallel composition as a product, relying on the interleaving semantics of concurrency. In this new transition system there is a variable $vc$ which ranges over sets of locations instead of a single one. The value of this variable is a subset of $L$ and it denotes all the locations in which control currently resides.

Note, however, that for SBS programs, the number of threads may grow unboundedly (we only model access to a shared resource, in principle it may be

---

2 Completeness is here understood relative to the expressibility of the first-order language.
accessed by many different components). Establishing a bound for these components is unnecessary and artificial. Instead, we construct a different transition system in which the transitions represent coarser atomic actions. This will be possible given certain peculiarities of SBS programs.

**Mutual exclusion:** as mentioned in Sec. 2 any process executing these programs begins with a P operation and ends with a V operation. Moreover, all the statements between them are executed in mutual exclusion. This is a characteristic of the method: split binary semaphores ensure mutual exclusion between any pair of P and V operations, i.e. at most one semaphore is on at any point of the execution. Mutual exclusion is therefore ensured.

From this property it follows that we can consider any sequence of the form $P_{s_i}; c_1; \cdots ; c_n; V_{s_j}$ as atomic. Such sequence will be called section $S^{ij}_p$. Execution of programs can be seen as the execution of a set $\{S^{ij}_1, \cdots , S^{ij}_n\}$ of such sections.

**Domino rule:** by the semaphore’s semantics we can assert that every V operation is followed by a corresponding P operation applied on the same semaphore, i.e. after a $S^{ij}_p$ ends, only a section $S^{jk}_q$ can begin.

**Locality of the variables:** the variables used in the SBS programs cannot be modified by other programs, i.e. the system is closed.

The mutual exclusion property implies that the sections $S^{ij}_p$ are executed one-at-a-time for each isolated thread. The possible interleavings among different threads are performed outside these sections, which can be considered atomic. Considering also the other two properties, the behavior of the system can be regarded as the execution of a sequence of sections $S^{ip_{j_1}j_2}_{p_{k_1}}; S^{ip_{j_2}j_3}_{p_{k_2}}; \cdots ; S^{ip_{j_n}j_{n+1}}_{p_{k_n}}$ with $j_{k_{n+1}} = i_{k_{n+1}}$. This gives these sort of systems a sequential flavor.

These characteristics allows us to define coarse grained transition systems which are much more suitable for simplification. These systems can be seen as sequential processes with non-deterministic jumps (goto sentences) between V’s and P’s applied to the same semaphore.

From these facts, each section $S^{ij}_p$ will be modeled as a transition and the locations identify which semaphore is active before and after the section is executed (i.e. $s_i$ and $s_j$ for the section $S^{ij}_p$). Loosely speaking, each transition will be associated with a sequence of actions executed inside a critical region by a thread.

In the general case, the SBS technique takes as input a set $\{S_0, \cdots , S_{m-1}\}$ of programs with its corresponding conditions $\{B_0, \cdots , B_{m-1}\}$, an initial condition $\Theta'$ and a global invariant $I$. Here, $m$ refers to the number of critical regions. For each program $S_i$ the technique generates a component SCC$_i$ implementing its corresponding critical region as shown in Fig. 3.

Only temporarily, we annotate each entry and exit point in a given component SCC$_i$ to identify the sections $S^{ij}_p$ (these labels will not identify nodes of the transition system): the entry points are marked with labels prefixed by “in” and the exit points are prefixed by “out”. The former are associated with P operations

---

3 For further discussions and an operational consideration, we refer to [4].
SCCi
\[ P.s_m ; (in_m) \]
if \( B_i \rightarrow \) skip
\[ \text{□} \neg B_i \rightarrow b_i := b_i + 1 ; \]
\[ V.s_m ; (out'_m) \]
\[ P.s_i ; (in_i) \]
\[ b_i := b_i - 1 \]
\[ \text{fi} \]
\[ S_i ; \]
if \( B_0 \land b_0 > 0 \rightarrow V.s_0 \) (out_o)
\[ \vdots \]
\[ \vdots \]
\[ \text{□} B_{m-1} \land b_{m-1} > 0 \rightarrow V.s_{(m-1)} \) (out_{m-1})\]
\[ \text{□} \bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0 \rightarrow V.s_m \) (out_m)\]
\[ \text{fi} \]

Fig. 3. Component with labels

and the later with V operations. The subscripts of the labels indicate which is the semaphore involved. For example, a section \((S^m_p)\) starting with \((in_m)\) may end in \((out'_m)\) (if condition \(\neg B_i\) holds) or in one of the \((out_j)\), with \(j \in \{0, \cdots, m\}\) (if \(B_i\) holds). On the other hand, the sections \((S^j_p)\) with \(j \in \{0, \cdots, m\}\) starting in \((in_i)\) should end in one of the exit points \((out_j)\) (if its corresponding guard is on).

Given the shape of the guarded transition systems, we need to calculate the condition under which a trace can be executed. This can be done by propagating the classical weakest precondition transformer \([5]\) for every guard inside a section \(S^j_p\) to the beginning of the section. We enumerate all possible transitions:

- From \((in_m)\) to \((out'_m)\): Each transition is executed if the condition \(\neg B_i\) holds and the state changes only with the increment of \(b_i\). Hence, we have the following \(m\) transitions (one for each component SCCi):

\[ vc = s_m \land \neg B_i \quad \longrightarrow \quad b_i := b_i + 1 ; \quad vc := s_m \]
with \(i \in \{0, \cdots, m-1\}\).

- From \((in_m)\) to \((out_j)\): These transitions are executed if \(B_i\) holds and \(B_j \land b_j > 0\) holds in the final guarded command. To calculate \(\gamma\) we apply wp on the guard \(B_j \land b_j > 0\). We have then \(m \times m\) transitions as follows:

\[ vc = s_m \land B_i \land \text{wp} (S_i)(B_j \land b_j > 0) \quad \longrightarrow \quad S_i ; \quad vc := s_j \]
with \(i \in \{0, \cdots, m-1\}, \quad j \in \{0, \cdots, m-1\}\).

- From \((in_m)\) to \((out_m)\): These transitions execute the same program as before but only when the last guard of the final guarded command holds for each SCCi. To calculate the guard \(\gamma\) we apply again wp but this time on guard \(\bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0\):

\[ vc = s_m \land B_i \land \text{wp} (S_i)(\bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0) \quad \longrightarrow \quad S_i ; \quad vc := s_m \]
with \(i \in \{0, \cdots, m-1\}\).
From (in\textsubscript{i}) to (out\textsubscript{j}): These transitions are executed when \(B_j \land b_j > 0\) holds in the last guarded command. To calculate the guard for the transition we apply \(wp\), using as program the decrement of \(b_i\) followed by program \(S_i\), on the guard \(B_j \land b_j > 0\). We obtain then \(m \times m\) transitions as follows:

\[
v_c = s_i \land wp(b_i := b_i - 1; S_i)(B_j \land b_j > 0) \rightarrow b_i := b_i - 1; S_i; v_c := s_j
\]

with \(i \in \{0, \ldots, m-1\}, j \in \{0, \ldots, m-1\} \).

From (in\textsubscript{i}) to (out\textsubscript{m}): These transitions are analogous to the ones considered in the previous item, but are executed when the last guard of the final guarded command holds for each \(SCC_i\). We now apply \(wp\) on these guards:

\[
v_c = s_i \land wp(b_i := b_i - 1; S_i)(\bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0) \rightarrow b_i := b_i - 1; S_i; v_c := s_m
\]

with \(i \in \{0, \ldots, m-1\} \).

These transitions form the set \(T\) of a transition system \(S = (V, \Theta, T)\) that model the behavior of the SBS-based process. \(V\) is the set of program variables in \(\{S_0, \ldots, S_{m-1}\}\) adding the auxiliary variables \(\{b_0, \ldots, b_{m-1}\}\) and the control variable \(vc\). The initial condition \(\Theta\) is:

\[
\Theta : vc = s_m \land (\bigwedge_{0 \leq i < m} b_i = 0) \land \Theta'.
\]

The SBS technique also provides an inductive invariant for the system:

\[
\varphi_{SBS} : \bigwedge_{0 \leq i \leq m} vc = s_i \rightarrow \varphi_{s_i}, \tag{7}
\]

with

\[
\varphi_{s_i} : B_i \land b_i > 0 \land I, \quad 0 \leq i < m
\]

\[
\varphi_{s_m} : \bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0 \land I. \tag{8}
\]

As an example we show how to obtain the transition system corresponding to the Producer/Consumer example (Sec. 3). In Fig. 4 we enumerate the lines of the producer program (without any invariant simplification) generated by the SBS technique in order to identify all possible transitions.

From de P operation at line 0 we have four sections that ends at the V operations at lines 2, 5, 6 and 7. We model each one with a transition as in equation 4. The guards \(\gamma_T\) are obtained finding out the conditions that should hold in order to execute the corresponding section. For example, if the section that ends at line 2 is executed, then, at the beginning of this execution, the condition guard \(\neg n < N\) in line 2 must hold. Then, this condition is the guard \(\gamma_T\) for the first transition. Moreover, the simultaneous assignments of the transitions are obtained directly by calculation of the state change produced by the
Fig. 4. Producer component
and by calculating the result of the \( wp \) transformer we get

\[
vc = s \land n + 1 > 0 \land c > 0 \implies b, n := b - 1, n + 1; vc := t .
\]

The transitions from the consumer program are obtained analogously. Figures 5 and 6 show all the resulting transitions.

**Producer transitions:**

- \( \tau_1 : vc = m \land \neg n < N \implies b := b + 1; vc := m , \)
- \( \tau_2 : vc = m \land n + 1 < N \land b > 0 \implies n := n + 1; vc := s , \)
- \( \tau_3 : vc = m \land n < N \land n + 1 > 0 \land c > 0 \implies n := n + 1; vc := t , \)
- \( \tau_4 : vc = m \land n < N \land (\neg n + 1 < N \land b = 0) \land (\neg n + 1 > 0 \lor c = 0) \)
  \( \implies n := n + 1; vc := m , \)
- \( \tau_5 : vc = s \land n + 1 < N \land b - 1 > 0 \implies b, n := b - 1, n + 1; vc := s , \)
- \( \tau_6 : vc = s \land n + 1 > 0 \land c > 0 \implies b, n := b - 1, n + 1; vc := t , \)
- \( \tau_7 : vc = s \land (\neg n + 1 < N \land b - 1 = 0) \land (\neg n + 1 > 0 \lor c = 0) \)
  \( \implies b, n := b - 1, n + 1; vc := m \)

**Consumer transitions:**

- \( \tau_8 : vc = m \land \neg n > 0 \implies c := c + 1; vc := m , \)
- \( \tau_9 : vc = m \land n > 0 \land n - 1 < N \land b > 0 \implies n := n - 1; vc := s , \)
- \( \tau_{10} : vc = m \land n - 1 > 0 \land c > 0 \implies n := n - 1; vc := t , \)
- \( \tau_{11} : vc = m \land n > 0 \land (\neg n - 1 < N \land b = 0) \land (\neg n - 1 > 0 \lor c = 0) \)
  \( \implies n := n - 1; vc := m , \)
- \( \tau_{12} : vc = t \land n - 1 < N \land b > 0 \implies c, n := c - 1, n - 1; vc := s , \)
- \( \tau_{13} : vc = t \land n - 1 > 0 \land c - 1 > 0 \implies c, n := c - 1, n - 1; vc := t , \)
- \( \tau_{14} : vc = t \land (\neg n - 1 < N \land b = 0) \land (\neg n - 1 > 0 \lor c - 1 = 0) \)
  \( \implies c, n := c - 1, n - 1; vc := m \)

**Fig. 5.** Bounded Buffer transitions.

These transitions form the set \( T \) of a transition system \( S = (V, \Theta, T) \) which models the behaviour of the producer and consumer processes. \( V \) is the set of program variables \( \{n, N, b, c, vc\} \). The initial condition \( \Theta \) is \( vc = m \land b = 0 \land c = 0 \land \Theta' \), where \( \Theta' \) is the initial condition of the problem \( n = 0 \land N > 0 \).

The SBS technique also provides an inductive invariant for the system:

\[
\varphi_{SBS} : (vc = s \implies n < N \land b > 0 \land n \geq 0) \land
\quad (vc = t \implies n > 0 \land c > 0 \land n \leq N) \land
\quad (vc = m \implies (\neg n < N \land b = 0) \land (\neg n > 0 \lor c = 0) \land 0 \leq n \leq N) .
\]

**7 Automatic Refinement**

We describe the process that automatically performs the required simplifications. In particular, we focus on the elimination of some guards in the final conditional statement of each component.
To check the possibility of elimination of a given guard, we use the backward propagation technique on a candidate invariant $\phi$ (5) which models the impossibility of the execution of this guard while the correction of the critical regions is preserved.

For example, if we want to eliminate guard $B_k \land b_k > 0$ (for some $k \in \{0, \ldots, m-1\}$) in program SCC$_i$ (Fig. 3) we strengthen the invariant to ensure that any transition with source (in$_m$) and target (out$_k$) or with source (in$_i$) and target (out$_k$), is never executed. This is achieved by strengthening the inductive invariant $\varphi_{SBS}$ (7) with the propagation (using wp) of the negation of the guard for each program SCC$_i$:

$$
\phi : \varphi_{SBD} \land F_m \land F_i
$$

with

$$F_m : \text{vc} = s_m \land B_i \rightarrow \text{wp} (S_i)(\neg B_k \lor b_k = 0)$$

$$F_i : \text{vc} = s_i \rightarrow \text{wp} (b_i := b_i - 1; S_i)(\neg B_k \lor b_k = 0)$$

the strengthening of the transitions with source (in$_m$) and target (out$_k$), and with source (in$_i$) and target (out$_k$) respectively.

Since we have a finite number of locations, we represent the set $\mathcal{L}$ as $\{0, \ldots, m\}$. Furthermore, with (2), every assertion $\phi$ over the transition system can be represented by arrays indexed over $\mathcal{L}$. That is, a formula

$$\phi(\vec{x}, \text{vc}) = \bigwedge_{l \in \mathcal{L}} \text{vc} = l \rightarrow \phi(\vec{x}, l)$$

will be represented by $[\phi(\vec{x}, 0), \ldots, \phi(\vec{x}, m)]$.

Given the formula $\phi$ and a formula $\varphi$ (both represented as arrays) we implement the formula transformer $\mathcal{B}$:
function $B(\phi, T, \varphi)$
  for every $i \in \mathcal{L}$ do
    $T_i := \{ \tau \in T : \text{src}(\tau) = i \}$ :
    $\varphi[i] := \phi[i] \land \bigwedge_{\tau \in T} \text{wp}(\Phi_{\tau})(\varphi[\text{tgt}(\tau)])$ ;
    $\varphi[i] := \mathfrak{R}\text{-simplify}(\varphi[i])$ ;
  end for
  return $\varphi$ ;

In each iteration the function obtains $B(\varphi)$ calculating the set $\{ \tau \in T : \text{src}(\tau) = i \}$ for each location. The result of the function is calculated using the formula

$$\text{WP}(\Phi)(\varphi(\vec{x}, vc)) = \bigwedge_{s \in \mathcal{L}} vc = s \rightarrow \bigwedge_{\tau \in T \land \text{src}(\tau) = s} \text{wp}(\Phi_{\tau})(\varphi_{\text{tgt}(\tau)}(\vec{x}))$$

which is equivalent to (3).

The function $\mathfrak{R}\text{-simplify}$ performs simplifications in the theory $\mathfrak{R}$ and always returns equivalent formulae. We implement them using CVC Lite and Isabelle. These are described in Sec. 8.

Using the function $B$ we can implement the algorithm for the fixed point:

function backPropagation($\phi, T, k$)
  $\varphi := [\text{True}, \cdots, \text{True}]$ ;
  $i := 0$ ;
  while $i < k$ do
    if $\not\models \Theta \rightarrow B(\phi, T, \varphi)$ then
      return unsat($B(\phi, T, \varphi)$) ;
    else if $\models \varphi \rightarrow B(\phi, T, \varphi)$ then
      return converge($\varphi$) ;
    else
      $\varphi := B(\phi, T, \varphi)$ ;
      $i := i + 1$ ;
    end if
  end while
  return notconverge($\varphi$) ;

This function has a main loop which calculates the values $\varphi_i$ in (6). The variable $\varphi$ stores these values. If the value $B(\phi, T, \varphi)$ does not satisfy $\models \Theta \rightarrow B(\phi, T, \varphi)$, then the function returns the value unsat($B(\phi, T, \varphi)$). In this case the method cannot find an inductive invariant for the formula $\phi$. In the other case, the method checks if it has reached the fix point. If it is the case, it returns the value converge($\varphi$) which stores the fix point. We use CVC Lite in order to implement the verifications of these conditions. The main loop do a maximum of $k$ iteration with $k$ a parameter of the algorithm. If the function cannot find a result in $k$ iterations the program finish returning the value notconverge($\varphi$) which store the last calculated formula.
8 Method Optimizations

During the execution of the procedure described in Sec. 7, the size of the formulae grows in each iteration. This is due to substitutions in the calculation of WP (4) performed by the B transformer. In order to optimize the procedure, we try to keep the size of the formulae as short as possible. We carry out this task in two ways: by eliminating and simplifying transitions and the candidate invariant before the execution of the method and applying simplification strategies over the formulae obtained in the fix point calculation.

Transition Simplification

The transitions from (in$_m$) to (out$_j$) in program SCC$_i$ (Fig. 3) are

\[ vc = s_m \land B_i \land wp(S_i)(B_j \land b_j > 0) \rightarrow S_i ; vc := s_j \]

with \( i \in \{0, \cdots, m - 1\}, j \in \{0, \cdots, m - 1\} \). When \( i = j \) the transition is never executed. This is proven by showing that the term corresponding to this transition in (3) is weaker than the candidate invariant \( \phi \), and hence can be absorbed in the fix point calculation. Then, these transitions will be:

\[ vc = s_m \land B_i \land wp(S_i)(B_j \land b_j > 0) \rightarrow S_i ; vc := s_j \]

with \( i \in \{0, \cdots, m - 1\}, j \in \{0, \cdots, m - 1\} \) and \( i \neq j \). The proof is in appendix A.1.

We can eliminate other transitions considering the strength in the candidate invariant \( \phi \) (9). If we try to eliminate the guard \( B_k \land b_k > 0 \) in the program SCC$_i$, the transitions from (in$_m$) to (out$_k$) and from (in$_i$) to (out$_k$):

\[ vc = s_m \land B_i \land wp(S_i)(B_k \land b_k > 0) \rightarrow S_i ; vc := s_k \]

and

\[ vc = s_i \land wp(b_i := b_i - 1; S_i)(B_k \land b_k > 0) \rightarrow b_i := b_i - 1 ; S_i ; vc := s_k \]

can be eliminated. This fact is proven by showing that the terms generated by these transitions are absorbed in the fix point calculation. The proofs are in appendix A.2.

We can also simplify the transition going from (in$_m$) to (out$_m$):

\[ vc = s_m \land B_i \land wp(S_i)(\bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0) \rightarrow S_i ; vc := s_m \]

with \( i \in \{0, \cdots, m - 1\} \). These transitions are simplified by eliminating a conjunction term in the subformula \( \bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0 \):

\[ vc = s_m \land B_i \land wp(S_i)(\bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0) \rightarrow S_i ; vc := s_m \]

with \( i \in \{0, \cdots, m - 1\} \). The proof is in the appendix A.4.
Candidate Invariant Simplification

If we try to eliminate the guard \( B_i \land b_i > 0 \) in the program \( \text{SCC}_i \) (i.e. eliminating the guard that releases the same kind of process) the strengthening of \( F_m \) in (9) is not necessary. The proof is in appendix A.3.

Formulae Simplifications

We implement some simplification strategies over the formulae. The function \( \mathcal{R}\text{-simplify} \) (Sec. 7) implements these strategies over the formulae \( \varphi[i] \) in the algorithm. These are quantifier-free formulae due to substitution in (4). Hence we can store them in conjunctive normal form. If some term in the normalized formulae is weaker than the conjunction of the others, this term can be eliminated. The process is applied over all terms in the formula. Also, a similar tactic is applied over the disjunctions inside the conjunctions; if a subterm is stronger than the disjunction of other subterms (in the same conjunctive term), it can be eliminated. The testing of these implications is done with the CVC Lite Validity Checker.

Moreover, other simplifications are implemented using the Isabelle theorem prover. This tool implements default tactics of simplification and the function \( \mathcal{R}\text{-simplify} \) uses them in order to reduce the size of the formulae.

9 Examples

We test the method over some classical problems. In order to perform this task we develop a program prototype. Fig. 7 shows a schematic diagram of the process performed by this software.

The program consists of two modules: Transition Generator and Invariant Verificator. The first one takes as input a specification of the problem and returns a representation of the transition system with the candidate invariant. The input specification consists of a set \( \{S_0, \ldots, S_{m-1}\} \) of programs with its corresponding conditions \( \{B_0, \ldots, B_{m-1}\} \), an initial condition \( \Theta' \) a global invariant \( I \) (as described in Sec. 6) and the guard that is tested for elimination. The output of this module is a simplified (see Sec. 8) transition system, the initial condition \( \Theta \) and the candidate invariant. This output is the input of the Invariant Verifier Module which implements the algorithms described in Sec. 7. Both inputs and outputs of the programs are text files. In this section we show the input and the result for each example in a formatted form. The program and the specifications of the examples can be found at http://www.cs.famaf.unc.edu.ar/~damian/publications/sbdinv/programs/.

Bounded Buffer

Consider the example outlined in section 3. The specification is:
First we test the method without any guard elimination. This case is for testing purpose only (we do not attempt to eliminate any guard). After perform the system simplifications the set of transitions generated is showed in Fig. 8. In this figure the locations are represented by the semaphore index as the transitions generator do. Note that after to perform the transition simplifications we obtain fewer transitions than the developed in Sec. 6.

The candidate invariant is also automatic generated and is the same SBS invariant of the problem $\varphi_{SBS}$ because we do not attempt to do any guard elimination:

$$
\begin{align*}
\phi :\ & (vc = 0 \rightarrow 0 \leq n \land n < N \land b_0 > 0) \land \\
& (vc = 1 \rightarrow 0 < n \land n \leq N \land b_1 > 0) \land \\
& (vc = 2 \rightarrow (\neg n < N \lor b_0 = 0) \land (\neg n > 0 \lor b_1 = 0) \land 0 \leq n \land n \leq N).
\end{align*}
$$
Furthermore, the initial condition generated is:

\[ \Theta : vc = 2 \land n = 0 \land b_0 = 0 \land b_1 = 0 \land N > 0 . \]

After a few seconds the method converge in one iteration, returning the same invariant \( \phi \).

The second test we attempt to eliminate the producer’s guard \( n < N \land b_0 > 0 \). This guard release producer that wait in the semaphore. The resulting set of transitions is written in Fig. 9 and the candidate invariant is

\[ \phi : (vc = 0 \rightarrow 0 \leq n \land n < N \land b_0 > 0 \land (b_0 = 1 \lor n + 1 < N)) \land (vc = 1 \rightarrow 0 < n \land n \leq N \land b_1 > 0) \land (vc = 2 \rightarrow (\neg n < N \lor b_0 = 0) \land (\neg n > 0 \lor b_1 = 0) \land 0 \leq n \land n \leq N) . \]
After a few seconds the method converge in four iterations, returning the following inductive invariant:

$$\varphi : (vc = 0 \rightarrow 0 \leq n \land n < N \land b_0 > 0 \land (n < N - 1 \rightarrow b_0 = 1)$$

$$\land (n > 0 \rightarrow b_0 \leq 2 \lor b_1 \leq 1)) \land$$

$$(vc = 1 \rightarrow 0 < n \land n \leq N \land b_1 > 0) \land (n > 1 \rightarrow b_0 \leq 1 \lor b_1 = 1)$$

$$\land (n < N \rightarrow b_0 \leq 1)) \land$$

$$(vc = 2 \rightarrow (\neg n < N \lor b_0 = 0) \land (\neg n > 0 \lor b_1 = 0)$$

$$\land 0 \leq n \land n \leq N)$$

which is implied by the initial condition $\Theta$. Hence we can eliminate this guard.

We also try also to eliminate the consumer’s guard $n > 0 \land b_1 > 0$ with similar positive result.

For the rests of guards ($n > 0 \land b_1 > 0$ in producer and $n < N \land b_3 > 0$ in consumer) the method detects the unsatisfiability of the formula $\Theta \rightarrow B(\varphi, T, \varphi)$ in the first iteration.

Table 1 show the details of this results. The first and second columns indicates the program and guard that

<table>
<thead>
<tr>
<th>Program</th>
<th>Guard</th>
<th>#Trs. Result</th>
<th>Iterations</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>Producer</td>
<td>11 eliminated</td>
<td>5</td>
<td>4.16&quot;</td>
</tr>
<tr>
<td>Producer</td>
<td>Consumer</td>
<td>10 detects unsat</td>
<td>2</td>
<td>1.23&quot;</td>
</tr>
<tr>
<td>Consumer</td>
<td>Producer</td>
<td>10 detects unsat</td>
<td>3</td>
<td>2.60&quot;</td>
</tr>
<tr>
<td>Consumer</td>
<td>Consumer</td>
<td>11 eliminated</td>
<td>5</td>
<td>4.17&quot;</td>
</tr>
</tbody>
</table>

Table 1. Bounded Buffer Results.

**Greedy Bounded Buffer**

This example is a modified version of the bounded buffer problem, where the consumer is greedy and consume two elements instead of one. The specification is:

$$\Theta' : n = 0 \land N > 1$$

$$I : 0 \leq n \land n \leq N$$

<table>
<thead>
<tr>
<th>producer:</th>
<th>consumer:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0 : n &lt; N$</td>
<td>$B_1 : n &gt; 1$</td>
</tr>
<tr>
<td>$S_0 : n := n + 1$</td>
<td>$S_1 : n := n - 2$</td>
</tr>
</tbody>
</table>

For this example the only guard that can be removed is $n > 1 \land b_1 > 0$ in the consumer program. This guard release a consumer process waiting in his semaphore.
The transition system generated have 11 transitions and the point fix calculation converge in 6 iterations at the invariant:

\[ \varphi : (\nu c = 0 \rightarrow 0 \leq n \wedge 0 < b_0 \wedge n < N \]
\[ \quad \wedge (3 \leq n \rightarrow b_1 = 1 \vee -0 < b_1) \]
\[ \quad \wedge (2 + n < N \wedge 1 \leq n \rightarrow (0 < b_1 \rightarrow (0 < -2 + b_0 \rightarrow b_1 = 1))) \]
\[ \quad \wedge (3 + n < N \rightarrow (0 < b_1 \rightarrow b_1 = 1 \vee -0 < -3 + b_0)) \]
\[ \quad \wedge (n + 1 < N \wedge 2 \leq n \rightarrow (0 < b_1 \rightarrow b_1 = 1 \vee -0 < b_0 - 1))) \]
\[ \wedge (\nu c = 1 \rightarrow n \leq N \wedge 0 < b_1 \wedge 1 \leq n \]
\[ \quad \wedge (3 < n \rightarrow b_1 = 1) \]
\[ \quad \wedge (n < N \wedge 3 \leq n \rightarrow (0 < b_1 - 1 \rightarrow (b_1 = 2 \vee -0 < -2 + b_0))) \]
\[ \quad \wedge (1 + n < N \rightarrow (0 < b_1 - 1 \rightarrow (0 < -3 + b_0 \rightarrow b_1 = 2))) \]
\[ \wedge (\nu c = 2 \rightarrow n \leq N \wedge 0 \leq n \wedge (b_1 = 0 \vee -1 < n) \wedge (b_0 = 0 \vee -n < N)) \]

For the others guards the method finish detecting the unsatisfiability of the formula \( \Theta \rightarrow B(\phi, T, \varphi) \). Results are shown in table 2.

<table>
<thead>
<tr>
<th>Program</th>
<th>Guard</th>
<th>#Trs. Result</th>
<th>Iterations</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>Producer</td>
<td>11 detects unsat</td>
<td>6</td>
<td>12.78”</td>
</tr>
<tr>
<td>Producer</td>
<td>Consumer</td>
<td>10 detects unsat</td>
<td>3</td>
<td>16.78”</td>
</tr>
<tr>
<td>Consumer</td>
<td>Producer</td>
<td>10 detects unsat</td>
<td>4</td>
<td>12.47”</td>
</tr>
<tr>
<td>Consumer</td>
<td>Consumer</td>
<td>11 eliminated</td>
<td>7</td>
<td>26.62”</td>
</tr>
</tbody>
</table>

**Table 2. Greedy Bounded Buffer Results.**

**Greedy Bounded Buffer M**

This example is a modified version of the bounded buffer problem, where the consumer is greedy and consume \( m \) elements with \( m \) a constant program. The specification is:

\[
\Theta' : n = 0 \wedge m > 0 \wedge N \geq m
\]
\[ I : 0 \leq n \wedge n \leq N \]
\[ \text{producer:} \]
\[ B_0 : n < N \]
\[ S_0 : n := n + 1 \]
\[ \text{consumer:} \]
\[ B_1 : n \geq m \]
\[ S_1 : n := n - m \]

For this problem the only guard that can be removed is \( n > 1 \wedge b_1 > 0 \) in the consumer program. This guard release a consumer process waiting in his semaphore. The method cannot find this possibilty of simplification in \( k = 20 \) fix point iterations. For the remaining guards the method finish detecting the unsatisfiability of the formula \( \Theta \rightarrow B(\phi, T, \varphi) \). Results are shown in table 3.
<table>
<thead>
<tr>
<th>Program</th>
<th>Guard</th>
<th>#Trs. Result</th>
<th>Iterations</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer</td>
<td>Producer</td>
<td>11 detects unsat</td>
<td>6</td>
<td>19.90&quot;</td>
</tr>
<tr>
<td>Producer</td>
<td>Consumer</td>
<td>10 detects unsat</td>
<td>2</td>
<td>2.36&quot;</td>
</tr>
<tr>
<td>Consumer</td>
<td>Producer</td>
<td>10 detects unsat</td>
<td>3</td>
<td>4.53&quot;</td>
</tr>
<tr>
<td>Consumer</td>
<td>Consumer</td>
<td>does not converge</td>
<td>20</td>
<td>22850.91&quot;</td>
</tr>
</tbody>
</table>

**Table 3.** Greedy Bounded Buffer M Results.

**General Semaphore**

This example is a classic implementation of general semaphores with binary semaphores. The specification of this problem is:

\[
\begin{align*}
\Theta' &: x = 0 \\
I &: x \geq 0
\end{align*}
\]

<table>
<thead>
<tr>
<th>P.</th>
<th>V:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B_0 : x &gt; 0)</td>
<td>(B_1 : True)</td>
</tr>
<tr>
<td>(S_0 : x := x - 1)</td>
<td>(S_1 : x := x + 1)</td>
</tr>
</tbody>
</table>

In this problem only the guard \(x > 0 \land b_0 > 0\) cannot be eliminated. This guard is in the V program and releases processes executing P programs. The method finish detecting this impossibility of simplification. For the other guard the methods finish detecting the simplifications. The details of these results are in table 4

<table>
<thead>
<tr>
<th>Program</th>
<th>Guard</th>
<th>#Trs. Result</th>
<th>Iterations</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P</td>
<td>11 eliminated</td>
<td>5</td>
<td>2.48&quot;</td>
</tr>
<tr>
<td>P</td>
<td>V</td>
<td>10 eliminated</td>
<td>3</td>
<td>0.91&quot;</td>
</tr>
<tr>
<td>V</td>
<td>P</td>
<td>detects unsat</td>
<td>2</td>
<td>0.72&quot;</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
<td>eliminated</td>
<td>3</td>
<td>0.83&quot;</td>
</tr>
</tbody>
</table>

**Table 4.** General Semaphore Results.

**Readers and Writers**

The last example is the classical readers and writers problem. The specification of this problem is:
In order to achieve the simplifications we need to add two kinds of assertions to the programs. First, given the topology of the programs (i.e. every exit is preceded by an entry), we can add the preconditions of the specification. This means that all the transitions in the writer’s exit will have predicate $w > 0$ as extra preconditions and similarly for the reader’s exit with $r > 0$. The transition generator implements this feature strengthening the guards $\gamma_\tau$ (see Sec. 4) with the corresponding preconditions in these transitions. Furthermore, since the conditions for the execution of both exit procedures is always true, variables $b_2$ and $b_4$ will be invariantly equal to 0. Hence, this two predicates are added to the global invariant.

With these additions, the result of the execution of the guard strengthening process is summarized in table 5.

<table>
<thead>
<tr>
<th>Program (entrance)</th>
<th>Program (exit)</th>
<th>#Trs.</th>
<th>Result</th>
<th>Iterations</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writer</td>
<td>Writer</td>
<td>39</td>
<td>eliminated</td>
<td>2</td>
<td>7.80&quot;</td>
</tr>
<tr>
<td>Writer</td>
<td>Reader</td>
<td>38</td>
<td>eliminated</td>
<td>2</td>
<td>11.87&quot;</td>
</tr>
<tr>
<td>Writer</td>
<td>Reader</td>
<td>38</td>
<td>eliminated</td>
<td>3</td>
<td>15.35&quot;</td>
</tr>
<tr>
<td>Writer</td>
<td>Writer</td>
<td>38</td>
<td>unsat detected</td>
<td>3</td>
<td>7.82&quot;</td>
</tr>
<tr>
<td>Writer</td>
<td>Reader</td>
<td>39</td>
<td>eliminated</td>
<td>3</td>
<td>7.66&quot;</td>
</tr>
<tr>
<td>Writer</td>
<td>Reader</td>
<td>38</td>
<td>unsat detected</td>
<td>3</td>
<td>8.88&quot;</td>
</tr>
<tr>
<td>Reader</td>
<td>Writer</td>
<td>38</td>
<td>eliminated</td>
<td>4</td>
<td>12.05&quot;</td>
</tr>
<tr>
<td>Reader</td>
<td>Writer</td>
<td>39</td>
<td>unsat detected</td>
<td>5</td>
<td>19.22&quot;</td>
</tr>
<tr>
<td>Reader</td>
<td>Reader</td>
<td>38</td>
<td>eliminated</td>
<td>4</td>
<td>4.11&quot;</td>
</tr>
<tr>
<td>Reader</td>
<td>Writer</td>
<td>38</td>
<td>unsat detected</td>
<td>3</td>
<td>5.09&quot;</td>
</tr>
<tr>
<td>Reader</td>
<td>Writer</td>
<td>38</td>
<td>eliminated</td>
<td>3</td>
<td>8.90&quot;</td>
</tr>
<tr>
<td>Reader</td>
<td>Reader</td>
<td>38</td>
<td>eliminated</td>
<td>4</td>
<td>4.09&quot;</td>
</tr>
<tr>
<td>Reader</td>
<td>Reader</td>
<td>39</td>
<td>eliminated</td>
<td>3</td>
<td>3.75&quot;</td>
</tr>
</tbody>
</table>

Table 5. Reader and Writer results.
10 Conclusions and Further Work

Conditional critical regions are a high level design pattern for concurrent programming. Unfortunately, since they are expensive to implement, most programming languages do not provide them as primitives. The SBS technique allows a nice implementation which can be further improved by axiomatic methods. This work provides a tool for performing such simplifications automatically. Since many problems can be solved with conditional critical regions, this method may have a wide range of applications. We performed some experiments with classical concurrent programs and variants of them (e.g. readers and writers, bounded buffer where the consumer is greedy and consume more than one element).

The examples suggest that in many cases the strengthening process terminates. However, until now we were unable to prove a termination theorem which would be of great importance since in this case the simplification could even be implemented in a compiler. Given that SBS was early considered as a valid alternative for implementation of conditional critical regions and monitors [7], these improvements may give a much more efficient implementation of monitors with conditional wait, which are easier to handle than other kinds of monitors.

Whereas the transition systems considered are generated by a SBS program, many of these results can be extended to other kinds of programs in which each transition models an atomic sequence of actions.

Although it works only for a very specific case, this work can be seen as a step towards the use of theorem provers not only for a posteriori verification of systems, but also for their formal construction and optimization.

A Demonstration of Transition System Simplifications

In this section we prove the simplifications of the transition system described in the section 8.

A.1

First, we prove the transitions from (in<sub>m</sub>) to (out<sub>i</sub>) in the program SCC<sub>i</sub> can be eliminated.

From the equations 3 and 5 we obtain the formulas ϕ<sup>j</sup> in each iteration of the fix point (figure 6):

\[ \varphi^{j+1}(\bar{x}, vc) = \phi(\bar{x}, vc) \land \bigwedge_{\tau \in T} vc = src(\tau) \rightarrow wp(\tau, \varphi^{j}_{tgt(\tau)}) \]

Using the fact in equation 2 we can rewrite this fix point definition:

\[ \varphi^{j+1} = \bigwedge_{\tau \in T} vc = src(\tau) \rightarrow \phi_{src(\tau)} \land wp(\tau, \varphi^{j}_{tgt(\tau)}) \] (10)
The term in the conjunction of this formulae corresponding to the transition is:

\[ vc = m \to \phi_m \land (B_i \land wp. S_i. (B_i \land b_i > 0) \to wp. S_i. \varphi_i^j) \]

We prove that the candidate invariant \( \phi_m \) absorb the right hand side of this equation:

\[
\phi_m \to (B_i \land wp. S_i. (B_i \land b_i > 0) \to wp. S_i. \varphi_i^j) \\
\iff \{ \phi_m \to \varphi_m \text{ by construction of } \phi \text{ (equation 9) } \} \\
\varphi_m \to (B_i \land wp. S_i. (B_i \land b_i > 0) \to wp. S_i. \varphi_i^j) \\
\iff \{ \varphi_m \to \neg B_i \lor b_i = 0 \text{ by equation 8 } \} \\
\neg B_i \lor b_i = 0 \to (B_i \land wp. S_i. (B_i \land b_i > 0) \to wp. S_i. \varphi_i^j) \\
\equiv \{ wp \text{ is conjunctive, } S_i \text{ does not depend on } b_i \} \\
\neg B_i \lor b_i = 0 \to (B_i \land b_i > 0 \land wp. S_i. B_i \to wp. S_i. \varphi_i^j) \\
\equiv \{ \text{Predicate calculus} \} \\
\text{True}
\]

Hence the term corresponding to the transitions does not contributes to the fix point calculation and the transition can be eliminated.

**A.2**

If we try to eliminate the guard \( B_k \land b_k > 0 \), the resulting strength of the candidate invariant allow to remove the transitions starting from \( \text{in}_m \) to \( \text{out}_k \) and from \( \text{in}_i \) to \( \text{out}_k \).

In the first case, the term in the equation 10 corresponding to the transition is:

\[ vc = m \to \phi_m \land (B_i \land wp. S_i. (B_k \land b_k > 0) \to wp. S_i. \varphi_k^j) \]

We prove that the candidate invariant \( \phi_m \) absorb the right hand side of this implication:

\[
\phi_m \to (B_i \land wp. S_i. (B_k \land b_k > 0) \to wp. S_i. \varphi_k^j) \\
\iff \{ \phi_m \to \varphi_m \text{ by construction of } \phi \text{ (equation 9) } \} \\
(B_i \to wp. S_i. (\neg B_k \lor b_k = 0)) \to (B_i \land wp. S_i. (B_k \land b_k > 0) \to wp. S_i. \varphi_k^j) \\
\equiv \{ \text{Predicate calculus} \} \\
wp. S_i. (\neg B_k \lor b_k = 0) \land B_i \land wp. S_i. (B_k \land b_k > 0) \to wp. S_i. \varphi_k^j \\
\equiv \{ wp \text{ is conjunctive } \} \\
B_i \land wp. S_i. (\neg B_k \lor b_k = 0) \land B_k \land b_k > 0 \to wp. S_i. \varphi_k^j \\
\equiv \{ \text{Predicate calculus} \} \\
B_i \land wp. S_i. False \to wp. S_i. \varphi_k^j \\
\equiv \{ \text{Law of Exclude Miracle } \} \\
B_i \land False \to wp. S_i. \varphi_k^j \\
\equiv \{ \text{Predicate calculus} \} \\
\text{True}
\]
The proof for the other case is similar. The term in the equation 10 corresponding to the transition is:

\[ vc = i \rightarrow \phi_i \land (wp. (b_i := b_i - 1; S_i). (B_k \land b_k > 0) \rightarrow wp. (b_i := b_i - 1; S_i). \varphi_k^j) \]

We prove that the candidate invariant \( \phi_i \) absorb the right hand side of this implication:

\[
\phi_i \rightarrow (wp. (b_i := b_i - 1; S_i). (B_k \land b_k > 0) \rightarrow wp. (b_i := b_i - 1; S_i). \varphi_k^j)
\]

\[
\Leftarrow \{ \phi_i \rightarrow wp. (b_i := b_i - 1; S_i). (\neg B_k \lor b_k = 0) \text{ by equation 9 } \}
\]

\[
wp. (b_i := b_i - 1; S_i). (\neg B_k \lor b_k = 0) \rightarrow (wp. (b_i := b_i - 1; S_i). (B_k \land b_k > 0) \rightarrow wp. (b_i := b_i - 1; S_i). \varphi_k^j)
\]

\[
\equiv \{ \text{Predicate calculus} \}
\]

\[
wp. (b_i := b_i - 1; S_i). (\neg B_k \lor b_k = 0) \land wp. (b_i := b_i - 1; S_i). (B_k \land b_k > 0) \rightarrow wp. (b_i := b_i - 1; S_i). \varphi_k^j
\]

\[
\equiv \{ \text{Predicate calculus} \}
\]

\[
wp. (b_i := b_i - 1; S_i). \text{False} \rightarrow wp. (b_i := b_i - 1; S_i). \varphi_k^j
\]

\[
\equiv \{ \text{Predicate calculus} \}
\]

\[
\text{True}
\]

A.3

The next possible simplification is when we tried to eliminate the guard \( B_i \land b_i > 0 \) in a program SCC\( F \) (the guard that release the same kind of process). In this case \( \varphi_{m} \) is stronger than \( F_m \). Hence, by the equation 9, we can ignore this strength in the candidate invariant. The proof is at follows:

\[
\varphi_{m} \rightarrow F_m
\]

\[
\Leftarrow \{ \varphi_{m} \rightarrow (vc = m \rightarrow \neg B_i \lor b_i = 0) \text{ by equation 8, } F_m \text{ definition (equation 9) } \}
\]

\[
(vc = m \rightarrow \neg B_i \lor b_i = 0) \rightarrow (vc = m \land B_i \rightarrow wp. S_i. (\neg B_i \lor b_i = 0))
\]

\[
\equiv \{ \text{Predicate calculus} \}
\]

\[
vc = m \land b_i = 0 \land B_i \rightarrow wp. S_i. (\neg B_i \lor b_i = 0)
\]

\[
\equiv \{ \text{wp is disjunctive because } S_i \text{ is deterministic } \}
\]

\[
vc = m \land b_i = 0 \land B_i \rightarrow wp. S_i. (\neg B_i) \lor wp. S_i. b_i = 0
\]

\[
\equiv \{ S_i \text{ does not depend on } b_i \} \}
\]

\[
vc = m \land b_i = 0 \land B_i \rightarrow wp. S_i. (\neg B_i) \lor b_i = 0
\]

\[
\equiv \{ \text{Predicate calculus} \}
\]

\[
\text{True}
\]
A.4

We can also simplify the transitions going from \((\text{in}_m)\) to \((\text{out}_m)\):

\[
v_c = m \land B_i \land \wp. S_i. \left( \bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0 \right) \longrightarrow S_i; \; v_c := m,
\]

with \(i \in \{0, \ldots, m - 1\} \).

The term, in the equation 10, corresponding to the transition is:

\[
v_c = m \rightarrow \phi_m \land \left( B_i \land \wp. S_i. \left( \bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0 \right) \rightarrow \wp. S_i. \varphi_j^i \right)
\]

We show a simplified equivalent formula of the right hand side of this implication:

\[
\phi_m \land \left( B_i \land \wp. S_i. \left( \bigwedge_{0 \leq j \leq m} \neg B_j \lor b_j = 0 \right) \rightarrow \wp. S_i. \varphi_j^i \right)
\]

\[
\equiv \begin{cases} \wp \text{ is conjunctive} & \phi_m \land \left( B_i \land \wp. S_i. \left( \bigwedge_{0 \leq j < m} \neg B_j \lor b_j = 0 \right) \rightarrow \wp. S_i. \varphi_j^i \right) \\ \wp \text{ is disjunctive because } S_i \text{ is deterministic} & \phi_m \land \left( \neg B_i \lor \neg \wp. S_i. \left( \bigwedge_{0 \leq j \leq m} \neg B_j \lor b_j = 0 \right) \rightarrow \wp. S_i. \varphi_j^i \right) \end{cases}
\]

\[
\equiv \begin{cases} \phi_m \rightarrow \neg B_i \lor b_i = 0 \text{ by equations 8 and 9} & \phi_m \land \left( \neg B_i \lor b_i = 0 \right) \land \left( \neg B_i \lor \left( \wp. S_i. \left( \bigwedge_{0 \leq j \leq m} \neg B_j \lor b_j = 0 \right) \rightarrow \wp. S_i. \varphi_m^i \right) \right) \\ \phi_m \rightarrow \neg \wp. S_i. \left( \bigwedge_{0 \leq j \leq m} \neg B_j \lor b_j = 0 \right) \rightarrow \wp. S_i. \varphi_j^i \text{ by equations 8 and 9} & \phi_m \land \left( \neg B_i \lor \left( \wp. S_i. \left( \bigwedge_{0 \leq j \leq m} \neg B_j \lor b_j = 0 \right) \rightarrow \wp. S_i. \varphi_j^i \right) \right) \end{cases}
\]

Hence, the simplified transitions are:

\[
v_c = m \land B_i \land \wp. S_i. \left( \bigwedge_{0 \leq j \leq m} \neg B_j \lor b_j = 0 \right) \longrightarrow S_i; \; v_c := m,
\]

with \(i \in \{0, \ldots, m - 1\} \).
References


