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 presents 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 $\varphi_{SBS} : \langle \forall s :: s = 0 \vee I_s \rangle$ 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 $\mathrm{S_o}$, $\mathrm{S_1}$ atomically under conditions B_0, B_1 respectively. Furthermore, the critical regions must preserve a global invariant 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.

$$\begin{aligned}\varphi_{SBS}: \; & (s_0 = 0 \vee (B_0 \wedge 0 < b_0 \wedge I)) \wedge \\ & (s_1 = 0 \vee (B_1 \wedge 0 < b_1 \wedge I)) \wedge \\ & (m = 0 \vee ((\neg B_0 \vee 0 = b_0) \wedge (\neg B_1 \vee 0 = b_1) \wedge I)) \;\; .\end{aligned}$$

Fig. 1 shows the (fully annotated) component obtained by the application of the SBS technique for the atomic execution of $\mathrm{S_o}$.

$\underline{\mathrm{SCC_o}}$
P.m
$\{I \wedge (\neg B_0 \vee b_0 = 0) \wedge (\neg B_1 \vee b_1 = 0)\}$
$b_0 := b_0 + 1$
$\{I \wedge (\neg B_0 \vee b_0 = 1) \wedge (\neg B_1 \vee b_1 = 0 \wedge b_0 > 0\}$
$\underline{\text{if}}$ $B_0 \rightarrow \{I \wedge B_0 \wedge b_0 > 0\}$ V.s_0
$\square \neg B_0 \rightarrow \{I \wedge (\neg B_0 \vee b_o = 0) \wedge (\neg B_1 \vee b_1 = 0)\}$ V.m
$\underline{\text{fi}}$
P.s_0
$\{I \wedge B_0 \wedge b_0 > 0\}$
$b_0 := b_0 - 1$
$\{I \wedge B_0\}$
$\mathrm{S_o}$
$\{I\}$
$\underline{\text{if}}$ $B_0 \wedge b_0 > 0 \rightarrow \{I \wedge B_0 \wedge b_0 > 0\}$ V.s_0
$\square$ $B_1 \wedge b_1 > 0 \rightarrow \{I \wedge B_1 \wedge b_1 > 0\}$ V.s_1
$\square (\neg B_0 \vee b_0 = 0) \wedge (\neg B_1 \vee b_1 = 0) \rightarrow$
$\{I \wedge (\neg B_0 \vee b_o = 0) \wedge (\neg B_1 \vee b_1 = 0)\}$ V.m
$\underline{\text{fi}}$

Fig. 1. SBS generated component

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.

$\underline{\mathrm{SCC_o}}$
P.m ;
$\{I \wedge (\neg B_0 \vee b_0 = 0) \wedge (\neg B_1 \vee b_1 = 0)\}$
$\underline{\text{if}}$ $B_0 \rightarrow \{I \wedge B_0 \wedge b_0 = 0 \wedge (\neg B_1 \vee b_1 = 0)\}$
skip
$\square$ $\neg B_0 \rightarrow \{I \wedge \neg B_0 \wedge (\neg B_1 \vee b_1 = 0)\}$
$b_0 := b_0 + 1$;
$\{I \wedge (\neg B_0 \vee b_0 = 0) \wedge (\neg B_1 \vee b_1 = 0)\}$
V.m ;
P.s_0 ;
$\{I \wedge B_0 \wedge b_0 > 0\}$
$b_0 := b_0 - 1$
$\underline{\text{fi}}$;
$\{I \wedge B_0\}$
$\mathrm{S_o}$;
$\{I\}$
$\underline{\text{if}}$ $B_0 \wedge b_0 > 0 \rightarrow \{I \wedge B_0 \wedge b_0 > 0\}$ V.s_0
$\square$ $B_1 \wedge b_1 > 0 \rightarrow \{I \wedge B_1 \wedge b_1 > 0\}$ V.s_1
$\square$ $(\neg B_0 \vee b_0 = 0) \wedge (\neg B_1 \vee b_1 = 0)$
$\rightarrow \{I \wedge (\neg B_0 \vee b_0 = 0) \wedge (\neg B_1 \vee b_1 = 0)\}$ V.m
$\underline{\text{fi}}$

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 *Producer/Consumer* problem through a bounded buffer. The component called *producer* produces some elements and send them to the *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.

Pre: $p = 0 \wedge d = 0$	
Prod: $*[$ produce.n ; $\{?\ p - d < N\}$; $buf.(p\ mod\ N) := n$; $p := p + 1$ $]$	Cons: $*[$ $\{?\ 0 < p - d\}$; $m := buf.(d\ mod\ N)$; $d := d + 1$; consume.m $]$

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

Pre: $n = 0$	
Prod: $*[\ \{?\ n < N\}$ $\quad ; n := n + 1$ $\quad]$	Cons: $*[\ \{?\ 0 < n\}$ $\quad ; n := n - 1$ $\quad]$

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:

$$\begin{aligned} \varphi_{SBS} : \ & (s = 0 \ \vee (n < N \wedge 0 < b)) \wedge \\ & (t = 0 \ \vee (0 < n \wedge 0 < c)) \wedge \\ & (m = 0 \vee ((n = N \vee 0 = b) \wedge (0 = n \vee 0 = c))) \ . \end{aligned}$$

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

Prod

P.m;
if $n < N \rightarrow$ skip
$\Box\ n = N \rightarrow b := b + 1$; V.$m$;
P.s ; $b := b - 1$
fi;
$n := n + 1$;
if $n < N \wedge 0 < b \rightarrow$ V.s
$\Box\ 0 < n \wedge 0 < c \ \rightarrow$ V.t
$\Box\ (n = N \vee 0 = b) \wedge$
$(0 = n \vee 0 = c) \rightarrow$ V.m
fi

Cons

P.m;
if $0 < n \rightarrow$ skip
$\Box\ 0 = n \rightarrow c := c + 1$; V.$m$;
P.t ; $c := c - 1$
fi;
$n := n - 1$;
if $n < N \wedge 0 < b \rightarrow$ V.s
$\Box\ 0 < n \wedge 0 < c \ \rightarrow$ V.t
$\Box\ (n = N \vee 0 = b) \wedge$
$(0 = n \vee 0 = c) \rightarrow$ V.m
fi

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

P.m;
if $n < N \rightarrow$ skip
$\Box\ n = N \rightarrow b := b + 1$; V.$m$; P.s ; $b := b - 1$
fi;
$n := n + 1$;
$\{0 < n?\}$
if $n < N \wedge 0 < b \rightarrow$ V.s
$\Box\ 0 < n \wedge 0 < c \rightarrow$ V.t
$\Box\ (n = N \vee 0 = b) \wedge (0 = n \vee 0 = c) \rightarrow$ V.m
fi

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.

Prod

P.m;
if $n < N \rightarrow$ skip
$\Box\ n = N \rightarrow b := b + 1$; V.$m$;
P.s ; $b := b - 1$
fi;
$n := n + 1$;
$\{0 < n\}$
if $n < N \wedge 0 < b \rightarrow$ V.s
$\Box\ 0 < c \rightarrow$ V.t
$\Box\ (n = N \vee 0 = b) \wedge 0 = c \rightarrow$ V.m
fi

Cons

P.m;
if $0 < n \rightarrow$ skip
$\Box\ 0 = n \rightarrow c := c + 1$; V.$m$;
P.t ; $c := c - 1$
fi;
$n := n - 1$;
$\{n < N\}$
if $0 < b \rightarrow$ V.s
$\Box\ 0 < n \wedge 0 < c \rightarrow$ V.t
$\Box\ 0 = b \wedge (0 = n \vee 0 = c) \rightarrow$ V.m
fi

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

P.m;
if $n < N \rightarrow$ skip
$\Box$ $n = N \rightarrow b := b + 1$; V.$m$; P.s ; $b := b - 1$
fi;
$n := n + 1$;
$\{0 < n\}\{n = N \vee 0 = b?\}$
if $n < N \wedge 0 < b \rightarrow$ V.s
$\Box$ $0 < c \rightarrow$ V.t
$\Box$ $(n = N \vee 0 = b) \wedge 0 = c \rightarrow$ V.m
fi

Taking the wp we can see that $n + 1 = N \vee 0 = b$ should hold after the first if. In the first branch it follows from $b = 0$. In the second branch we need $n + 1 = N \vee 1 = b$ as a valid postcondition for the P.s. This can be achieved by strengthening the invariant for s to $s = 0 \vee (0 < b \wedge n < N \wedge (n+1 = N \vee 1 = b))$. 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 wp we can see that the invariant for semaphore t has to be strengthened as well into $t = 0 \vee (0 < c \wedge 0 < n \wedge (b \leq 1 \vee n = N))$. Furthermore, the last guard can be simplified provided that the required condition holds.

The annotated program then becomes

Prod

P.m;
if $n < N \rightarrow$ skip
$\Box$ $n = N \rightarrow b := b + 1$; V.$m$;
P.s ; $b := b - 1$
fi;
$n := n + 1$;
$\{0 < n\}$
if $0 < c \rightarrow$ V.t
$\Box$ $0 = c \rightarrow$ V.m
fi

Cons

P.m;
if $0 < n \rightarrow$ skip
$\Box$ $0 = n \rightarrow c := c + 1$; V.$m$;
P.t ; $c := c - 1$
fi;
$n := n - 1$;
$\{n < N\}$
if $0 < b \rightarrow$ V.s
$\Box$ $0 < n \wedge 0 < c \rightarrow$
$\{b \leq 1 \vee n = N?\}$ V.t
$\Box$ $0 = b \wedge (0 = n \vee 0 = c) \rightarrow$ V.m
fi

The remaining proof obligation can be ignored since we plan to eliminate this branch of the if altogether. This is easily done by ensuring the condition $0 = n \vee 0 = c$ prior to the last 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:

$$s = 0 \vee 0 < b \wedge n < N \wedge (n + 1 = N \vee 1 = b) \wedge (c \leq 1 \vee 0 = n)$$
$$t = 0 \vee 0 < c \wedge 0 < n \wedge (1 = n \vee 1 = c) \wedge (b \leq 1 \vee n = N)$$

whereas the whole program (without annotations) becomes

Prod	Cons
P.m;	P.m;
if $n < N \rightarrow$ skip	if $0 < n \rightarrow$ skip
$\Box$ $n = N \rightarrow b := b + 1$; V.$m$;	$\Box$ $0 = n \rightarrow c := c + 1$; V.$m$;
P.s ; $b := b - 1$	P.t ; $c := c - 1$
fi;	fi;
$n := n + 1$;	$n := n - 1$;
if $0 < c \rightarrow$ V.t	if $0 < b \rightarrow$ V.s
$\Box$ $0 = c \rightarrow$ V.m	$\Box$ $0 = b \rightarrow$ V.m
fi	fi

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 Σ 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 Σ. 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* $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ 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:

$$\gamma_\tau \ \longmapsto \ \bar{x} := \bar{e}_\tau(\bar{x})$$

where $\gamma_\tau \in \mathcal{F}$ is the *transition guard* and $\bar{e}_\tau(\bar{x})$ is a sequence of expressions in Σ whose free variables are taken from $\mathcal{V}$; both sequences should have equal length. The formula γ_τ 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:

$$vc = l_\tau \wedge \gamma_\tau \ \longmapsto \ \bar{x} := \bar{e}_\tau(\bar{x}) \, ; vc := l'_\tau$$

where $l_\tau, l'_\tau \in \mathcal{L}$, γ_τ 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 τ we define the functions $src(\tau) = l_\tau$ and $tgt(\tau) = l'_\tau$.

For each transition, we define a predicate Φ_τ as follows:

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

where $\bar{x}'$ is the renaming of variables x_i in $\bar{x}$ as x'_i. Note that Φ_τ 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* Φ of a guarded transition system $\mathcal{S}$ as follows:

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

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

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

The universal quantifier can be eliminated using substitution:

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

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

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

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

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

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:

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

or, using (1):

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

5 Invariants

Let $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ be a guarded transition system and $\mathfrak{R}$ a first-order theory over the language Σ. A formula $\varphi \in \mathcal{F}$ is an *inductive invariant* if $\mathfrak{R} \models \Theta \rightarrow \varphi$ and $\mathfrak{R} \models \varphi \rightarrow \mathrm{WP}\,(\Phi)(\varphi)$, with Φ 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 φ such that $\models \varphi \rightarrow \phi$. This characterization is sound and relative complete[2].

For a monotone formula transformer $\Gamma : \mathcal{F} \mapsto \mathcal{F}$, such as WP, we write the greatest fixed point as $\nu X.\Gamma(X)$. Its meaning is the usual [3].

Given a formula ϕ, we define the monotone formula transformer $\mathcal{B}$ by

$$\mathcal{B}(Y) \triangleq \phi \wedge \mathrm{WP}\,(\Phi)(Y) \ . \tag{5}$$

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

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

$$\underbrace{\mathit{True}}_{\varphi^0} \leftarrow \underbrace{\mathcal{B}(\varphi_0)}_{\varphi^1} \leftarrow \underbrace{\mathcal{B}(\varphi_1)}_{\varphi^2} \leftarrow \cdots \tag{6}$$

converges in finitely many steps, then its limit is $\varphi_{\mathcal{B}}$. From this property we can explore the state space using the *backward propagation technique* [3]: given a candidate invariant ϕ, we can find the fixed point $\varphi_{\mathcal{B}}$ if the sequence converges in finitely many steps. Then, proving $\models \Theta \rightarrow \varphi_{\mathcal{B}}$ we can verify if ϕ 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 $\mathcal{L}$ 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[3].
From this property it follows that we can consider any sequence of the form $\mathrm{P}.s_i; c_1; \cdots; c_n; \mathrm{V}.s_j$ as atomic. Such sequence will be called section S_p^{ij}. Execution of programs can be seen as the execution of a set $\{S_1^{i_1 j_1}, \cdots, S_n^{i_n j_n}\}$ 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_p^{ij} ends, only a section S_q^{jk} 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_p^{ij} 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 $S_{p_1}^{i_{p_1} j_{p_1}}; S_{p_2}^{i_{p_2} j_{p_2}}; \cdots; S_{p_n}^{i_{p_n} j_{p_n}}; \cdots$ with $j_{p_k} = i_{p_{k+1}}$. 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_p^{ij} 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_p^{ij}). 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 $\{\mathrm{S}_0, \cdots, \mathrm{S}_{m-1}\}$ of programs with its corresponding conditions $\{B_0, \cdots, B_{m-1}\}$, an initial condition Θ' 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_p^{jk} (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].

$\underline{\text{SCC}_i}$

P.s_m ; (in$_m$)
<u>if</u> B_i $\rightarrow$ skip
$\Box$ $\neg B_i$ $\rightarrow$ $b_i := b_i + 1$;
V.s_m ; (out'$_m$)
P.s_i ; (in$_i$)
$b_i := b_i - 1$
<u>fi</u>;
S$_i$;

<u>if</u> $B_0 \wedge b_0 > 0$	$\rightarrow$	V.s_0	(out$_0$)
$\vdots$		$\vdots$	$\vdots$
$\Box$ $B_{m-1} \wedge b_{m-1} > 0$	$\rightarrow$	V.$s_{(m-1)}$	(out$_{m-1}$)
$\Box$ $\bigwedge_{0 \leq j < m} \neg B_j \vee b_j = 0$	$\rightarrow$	V.s_m	(out$_m$)

<u>fi</u>

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_p^{mm}) 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_p^{ij} 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_p^{jk} 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 SCC$_i$):

$$vc = s_m \wedge \neg B_i \;\longmapsto\; b_i := b_i + 1 \,;\, vc := s_m$$

  with $i \in \{0, \cdots, m-1\}$.
- From (in$_m$) to (out$_j$): These transitions are executed if B_i holds and $B_j \wedge b_j > 0$ holds in the final guarded command. To calculate γ we apply wp on the guard $B_j \wedge b_j > 0$. We have then $m \times m$ transitions as follows:

$$vc = s_m \wedge B_i \wedge \text{wp}\,(S_i)(B_j \wedge b_j > 0) \;\longmapsto\; S_i \,;\, vc := s_j$$

  with $i \in \{0, \cdots, m-1\}$, $j \in \{0, \cdots, m-1\}$.
- 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 SCC$_i$. To calculate the guard γ we apply again wp but this time on guard $\bigwedge_{0 \leq j < m} \neg B_j \vee b_j = 0$:

$$vc = s_m \wedge B_i \wedge \text{wp}\,(S_i)(\textstyle\bigwedge_{0 \leq j < m} \neg B_j \vee b_j = 0) \;\longmapsto\; S_i \,;\, vc := s_m$$

  with $i \in \{0, \cdots, m-1\}$.

– From (in_i) to (out_j): These transitions are executed when $B_j \wedge b_j > 0$ 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 $B_j \wedge b_j > 0$. We obtain then $m \times m$ transitions as follows:

$$\begin{array}{c} vc = s_i \wedge \text{wp}\,(b_i := b_i - 1; S_i)(B_j \wedge b_j > 0) \\ \longmapsto\ b_i := b_i - 1\,;S_i\,;vc := s_j \end{array}$$

with $i \in \{0, \cdots, m-1\}$, $j \in \{0, \cdots, m-1\}$.

– From (in_i) to (out_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:

$$\begin{array}{c} vc = s_i \wedge \text{wp}\,(b_i := b_i - 1; S_i)(\bigwedge_{0 \le j < m} \neg B_j \vee b_j = 0) \\ \longmapsto\ b_i := b_i - 1\,;S_i\,;vc := s_m \end{array}$$

with $i \in \{0, \cdots, m-1\}$.

These transitions form the set $\mathcal{T}$ of a transition system $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ that model the behavior of the SBS-based process. $\mathcal{V}$ is the set of program variables in $\{\text{S}_0, \cdots, \text{S}_{m-1}\}$ adding the auxiliary variables $\{b_0, \cdots, b_{m-1}\}$ and the control variable vc. The initial condition Θ is:

$$\Theta : vc = s_m \wedge (\textstyle\bigwedge_{0 \le i < m} b_i = 0) \wedge \Theta' \ .$$

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

$$\varphi_{SBS} : \bigwedge_{0 \le i \le m} vc = s_i \rightarrow \varphi_{s_i} \tag{7}$$

with

$$\begin{array}{l} \varphi_{s_i} : B_i \wedge b_i > 0 \wedge I \qquad , 0 \le i < m \\ \varphi_{s_m} : (\bigwedge_{0 \le j < m} \neg B_j \vee b_j = 0) \wedge I \ . \end{array} \tag{8}$$

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 γ_τ 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 begining of this execution, the condition guard $\neg n < N$ in line 2 must hold. Then, this condition is the guard γ_τ for the first transition. Moreover, the simultaneous assignements of the transitions are obtained directly by calculation of the state change produced by the

Prod

0: P.m;
1: if $n < N \rightarrow$ skip
2: $\square\ \neg n < N \rightarrow b := b + 1$; V.$m$; P.s ; $b := b - 1$
3: fi;
4: $n := n + 1$;
5: if $n < N \wedge b > 0 \rightarrow$ V.s
6: $\square\ n > 0 \wedge c > 0 \rightarrow$ V.t
7: $\square\ (\neg n < N \vee b = 0) \wedge (\neg n > 0 \vee c = 0) \rightarrow$ V.m
8: fi

Fig. 4. Producer component

correspondig section, and the locations are represented by the initial semaphore in the P operation and the final one in the ending V operation. In this case the only assignement is $b := b+1$ and the initial and final locations is the semaphore m . Then, the transition is:

$$vc = m \wedge \neg n < N \longmapsto b := b + 1; vc := m \ .$$

We follow with the three next sections begining at line 0. In order to execute them the guard $n < N$ in the line 1 must hold. Moreover, each section ends by executing one test at the final if sentence (line 5). The transitions corresponding to each of the alternatives should be executed only if its final condition holds at the end of the execution. Since the definition of the transition systems require that the guard for each transition is known in advance, we back propagate them with the classical *weakest precondition* transformer ([5]) over the section. The simultaneous assignements of the transitions is the sentence in the line 4 and the locations are the semaphores involved in each trace. For example, for the trace that end at the V operation in the line 5 the transitions is:

$$vc = m \wedge n < N \wedge wp.\, n := n + 1.\, (n < N \wedge b > 0) \longmapsto n := n + 1; vc := s \ .$$

Calculating the result of the wp transformer and simplificating the guard, this transitions can be writed as:

$$vc = m \wedge n + 1 < N \wedge b > 0 \longmapsto n := n + 1; vc := s \ .$$

The rest of transitions starting at line 0 are calculated in the same way.

There are also three sections starting at the P operation in line 2 an ending at the V operations in lines 5, 6 and 7. Each one is represent by transitions begining at location s and ending at locations s, t and m. The guard part of these traces are obtained back propagating the guards of the final if sentence as described above. These traces execute the assignements in the lines 2 and 4 and will be modelled as a single simultaneous assignement. For example, the trace ending at line 6 is modelled as the folowing transition:

$$vc = s \wedge wp.\, b, n := b - 1, n + 1.\, (n > 0 \wedge c > 0) \longmapsto b, n := b - 1, n + 1; vc := t$$

and by calculating the result of the *wp* transformer we get

$$vc = s \wedge n + 1 > 0 \wedge c > 0 \longmapsto b, n := b - 1, n + 1; vc := t \ .$$

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

Producer transitions:

$$\begin{aligned}
&\tau_1 : vc = m \wedge \neg n < N \longmapsto b := b + 1; vc := m \ , \\
&\tau_2 : vc = m \wedge n + 1 < N \wedge b > 0 \longmapsto n := n + 1; vc := s \ , \\
&\tau_3 : vc = m \wedge n < N \wedge n + 1 > 0 \wedge c > 0 \longmapsto n := n + 1; vc := t \ , \\
&\tau_4 : vc = m \wedge n < N \wedge (\neg n + 1 < N \vee b = 0) \wedge (\neg n + 1 > 0 \vee c = 0) \\
&\qquad \longmapsto n := n + 1; vc := m \ , \\
&\tau_5 : vc = s \wedge n + 1 < N \wedge b - 1 > 0 \longmapsto b, n := b - 1, n + 1; vc := s \ , \\
&\tau_6 : vc = s \wedge n + 1 > 0 \wedge c > 0 \longmapsto b, n := b - 1, n + 1; vc := t \ , \\
&\tau_7 : vc = s \wedge (\neg n + 1 < N \vee b - 1 = 0) \wedge (\neg n + 1 > 0 \vee c = 0) \\
&\qquad \longmapsto b, n := b - 1, n + 1; vc := m
\end{aligned}$$

Consumer transitions:

$$\begin{aligned}
&\tau_8 : \ vc = m \wedge \neg n > 0 \longmapsto c := c + 1; vc := m \ , \\
&\tau_9 : \ vc = m \wedge n > 0 \wedge n - 1 < N \wedge b > 0 \longmapsto n := n - 1; vc := s \ , \\
&\tau_{10} : vc = m \wedge n - 1 > 0 \wedge c > 0 \longmapsto n := n - 1; vc := t \ , \\
&\tau_{11} : vc = m \wedge n > 0 \wedge (\neg n - 1 < N \vee b = 0) \wedge (\neg n - 1 > 0 \vee c = 0) \\
&\qquad \longmapsto n := n - 1; vc := m \ , \\
&\tau_{12} : vc = t \wedge n - 1 < N \wedge b > 0 \longmapsto c, n := c - 1, n - 1; vc := s \ , \\
&\tau_{13} : vc = t \wedge n - 1 > 0 \wedge c - 1 > 0 \longmapsto c, n := c - 1, n - 1; vc := t \ , \\
&\tau_{14} : vc = t \wedge (\neg n - 1 < N \vee b = 0) \wedge (\neg n - 1 > 0 \vee c - 1 = 0) \\
&\qquad \longmapsto c, n := c - 1, n - 1; vc := m
\end{aligned}$$

Fig. 5. Bounded Buffer transitions.

These transitions form the set $\mathcal{T}$ of a transition system $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{T} \rangle$ which models the behavoir of the producer and consumer processes. $\mathcal{V}$ is the set of program variables $\{n, N, b, c, vc\}$. The initial condition Θ is $vc = m \wedge b = 0 \wedge c = 0 \wedge \Theta'$, where Θ' is the initial condition of the problem $n = 0 \wedge N > 0$.

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

$$\begin{aligned}
\varphi_{SBS} : \ &(vc = s \ \ \rightarrow n < N \wedge b > 0 \wedge n \geq 0) \wedge \\
&(vc = t \ \ \rightarrow n > 0 \wedge c > 0 \wedge n \leq N) \wedge \\
&(vc = m \rightarrow (\neg n < N \vee b = 0) \wedge (\neg n > 0 \vee c = 0) \wedge 0 \leq n \leq N) \ .
\end{aligned}$$

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.

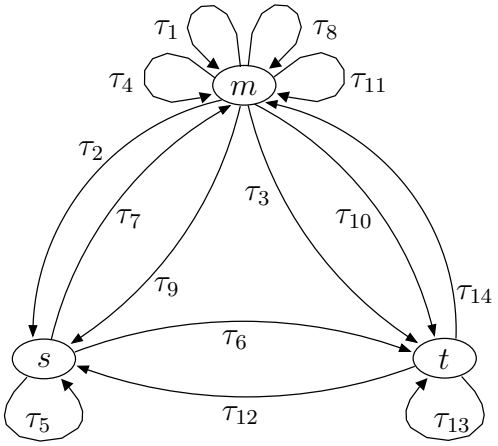

Fig. 6. Bounded Buffer diagram.

To check the possibility of elimination of a given guard, we use the backward propagation technique on a candidate invariant ϕ (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 $B_k \wedge b_k > 0$ (for some $k \in \{0, \cdots, m-1\}$) 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 φ_{SBS} (7) with the propagation (using wp) of the negation of the guard for each program SCC_i:

$$\phi : \varphi_{SBD} \wedge F_m \wedge F_i \tag{9}$$

with $\quad F_m : vc = s_m \wedge B_i \;\rightarrow\; \text{wp}\,(S_i)(\neg B_k \vee b_k = 0)$
$\quad F_i \;\; : vc = s_i \;\rightarrow\; \text{wp}\,(b_i := b_i - 1; S_i)(\neg B_k \vee b_k = 0)$

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 $\{0, \cdots, m\}$. Furthermore, with (2), every assertion ϕ over the transition system can be represented by arrays indexed over $\mathcal{L}$. That is, a formula

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

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

Given the formula ϕ and a formula φ (both represented as arrays) we implement the formula transformer $\mathcal{B}$:

function $\mathcal{B}(\phi, \mathcal{T}, \varphi)$
  for every $i \in \mathcal{L}$ **do**
    $\mathcal{T}_i := \{\tau \in \mathcal{T} : src(\tau) = i\}$;
    $\varphi[i] := \phi[i] \ \wedge \ \bigwedge_{\tau \in \mathcal{T}_i} \mathrm{wp}\,(\Phi_\tau)(\varphi[tgt(\tau)])$;
    $\varphi[i] := \mathfrak{R}$-simplify$(\varphi[i])$;
  end for
  return φ ;

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

$$\mathrm{WP}\,(\Phi)(\varphi(\bar{x}, vc)) = \bigwedge_{s \in \mathcal{L}} vc = s \ \rightarrow \bigwedge_{\substack{\tau \in \mathcal{T} \wedge \\ src(\tau) = s}} \mathrm{wp}\,(\Phi_\tau)(\varphi_{tgt(\tau)}(\bar{x}))$$

which is equivalent to (3).

The function $\mathfrak{R}$-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 $\mathcal{B}$ we can implement the algorithm for the fixed point:

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

This function has a main loop which calculates the values φ_i in (6). The variable φ stores these values. If the value $\mathcal{B}(\phi, \mathcal{T}, \varphi)$ does not satisfy $\models \Theta \rightarrow \mathcal{B}(\phi, \mathcal{T}, \varphi)$, then the function returns the value unsat$(\mathcal{B}(\phi, \mathcal{T}, \varphi))$. In this case the method cannot find an inductive invariant for the formula ϕ. In the other case, the method checks if it has reached the fix point. If it is the case, it returns the value converge(φ) which stores the fix point. We use CVC Lite in order to implement the verifications of these conditions. The main loop do a maximun 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(φ) wich 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 $\mathcal{B}$ 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

$$vc = s_m \wedge B_i \wedge \text{wp}\,(S_i)(B_j \wedge b_j > 0) \;\longmapsto\; S_i\,; vc := s_j$$

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 ϕ, and hence can be absorbed in the fix point calculation. Then, these transitions will be:

$$vc = s_m \wedge B_i \wedge \text{wp}\,(S_i)(B_j \wedge b_j > 0) \;\longmapsto\; S_i\,; vc := s_j$$

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 ϕ (9). If we try to eliminate the guard $B_k \wedge b_k > 0$ in the program SCC_i, the transitions from (in_m) to (out_k) and from (in_i) to (out_k)

$$vc = s_m \wedge B_i \wedge \text{wp}\,(S_i)(B_k \wedge b_k > 0) \;\longmapsto\; S_i\,; vc := s_k \quad \text{and}$$
$$vc = s_i \wedge \text{wp}\,(b_i := b_i - 1; S_i)(B_k \wedge b_k > 0) \longmapsto\; b_i := b_i - 1\,; S_i\,; vc := s_k$$

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):

$$vc = s_m \wedge B_i \wedge \text{wp}\,(S_i)(\textstyle\bigwedge_{0 \leq j < m} \neg B_j \vee b_j = 0) \;\longmapsto\; S_i\,; vc := s_m$$

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

$$vc = s_m \wedge B_i \wedge \text{wp}\,(S_i)(\textstyle\bigwedge_{\substack{0 \leq j < m \\ i \neq j}} \neg B_j \vee b_j = 0) \;\longmapsto\; S_i\,; vc := s_m$$

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 $B_i \wedge b_i > 0$ in the program 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 $\mathfrak{R}$-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 $\mathfrak{R}$-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 $\{\mathrm{S}_0, \cdots, \mathrm{S}_{m-1}\}$ of programs with its corresponding conditions $\{B_0, \cdots, B_{m-1}\}$, an initial condition Θ' 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 Θ 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:

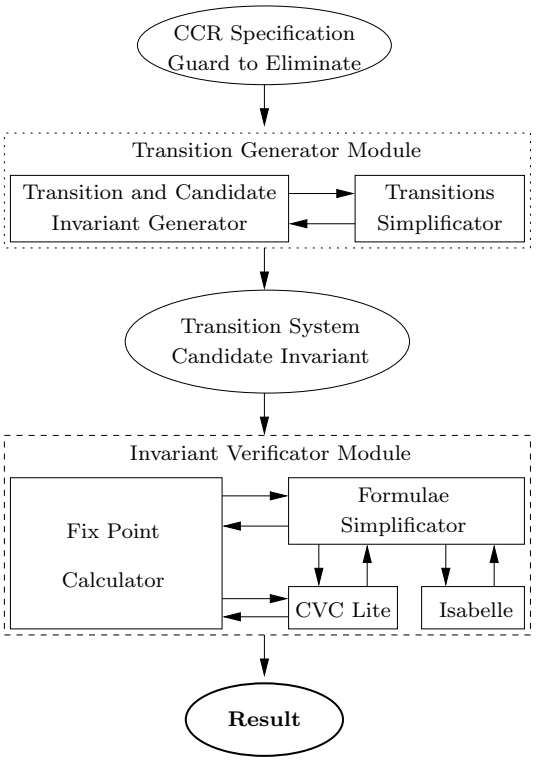

Fig. 7. Process Diagram.

$\Theta' : n = 0 \wedge N > 0$ $I : \ 0 \leq n \wedge n \leq N$	
producer: $B_0 : n < N$ $S_0 : n := n + 1$	consumer: $B_1 : n > 0$ $S_1 : n := n - 1$

First we test the method without any guard elimination. This case is for testing pourpose only (we do not attempt to eliminate any guard). After perform the system simplificactions 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 φ_{SBS} because we do not attempt to do any guard elimination:

$$\begin{aligned}\phi : &(vc = 0 \rightarrow 0 \leq n \wedge n < N \wedge b_0 > 0) \wedge \\ &(vc = 1 \rightarrow 0 < n \wedge n \leq N \wedge b_1 > 0) \wedge \\ &(vc = 2 \rightarrow (\neg n < N \vee b_0 = 0) \wedge (\neg n > 0 \vee b_1 = 0) \wedge 0 \leq n \wedge n \leq N) \ .\end{aligned}$$

$$
\begin{aligned}
&vc = 2 \wedge \neg n < N \longmapsto b_0 := b_0 + 1; vc := 2\ ,\\
&vc = 2 \wedge \neg n > 0 \longmapsto b_1 := b_1 + 1; vc := 2\ ,\\
&vc = 2 \wedge n < N \wedge b_1 > 0 \wedge n + 1 > 0 \longmapsto n := n + 1; vc := 1\ ,\\
&vc = 2 \wedge n > 0 \wedge b_0 > 0 \wedge n - 1 < N \longmapsto n := n - 1; vc := 0\ ,\\
&vc = 2 \wedge n < N \wedge (b_1 = 0 \vee \neg n + 1 > 0) \longmapsto n := n + 1; vc := 2\ ,\\
&vc = 2 \wedge n > 0 \wedge (b_0 = 0 \vee \neg n - 1 < N) \longmapsto n := n - 1; vc := 2\ ,\\
&vc = 0 \wedge b_0 - 1 > 0 \wedge n + 1 < N \longmapsto b_0, n := b_0 - 1, n + 1; vc := 0\ ,\\
&vc = 0 \wedge b_1 > 0 \wedge n + 1 > 0 \longmapsto b_0, n := b_0 - 1, n + 1; vc := 1\ ,\\
&vc = 1 \wedge b_0 > 0 \wedge n - 1 < N \longmapsto b_1, n := b_1 - 1, n - 1; vc := 0\ ,\\
&vc = 1 \wedge b_1 - 1 > 0 \wedge n - 1 > 0 \longmapsto b_1, n := b_1 - 1, n - 1; vc := 1\ ,\\
&vc = 0 \wedge (b_0 - 1 = 0 \vee \neg n + 1 < N) \wedge (b_1 = 0 \vee \neg n + 1 > 0)\\
&\qquad\qquad \longmapsto b_0, n := b_0 - 1, n + 1; vc := 2\ ,\\
&vc = 1 \wedge (b_0 = 0 \vee \neg n - 1 < N) \wedge (b_1 - 1 = 0 \vee \neg n - 1 > 0)\\
&\qquad\qquad \longmapsto b_1, n := b_1 - 1, n - 1; vc := 2
\end{aligned}
$$

Fig. 8. Bounded Buffer transitions.

$$
\begin{aligned}
&vc = 2 \wedge \neg n < N \longmapsto b_0 := b_0 + 1; vc := 2\ ,\\
&vc = 2 \wedge \neg n > 0 \longmapsto b_1 := b_1 + 1; vc := 2\ ,\\
&vc = 2 \wedge n < N \wedge b_1 > 0 \wedge n + 1 > 0 \longmapsto n := n + 1; vc := 1\ ,\\
&vc = 2 \wedge n > 0 \wedge b_0 > 0 \wedge n - 1 < N \longmapsto n := n - 1; vc := 0\ ,\\
&vc = 2 \wedge n < N \wedge (b_1 = 0 \vee \neg n + 1 > 0) \longmapsto n := n + 1; vc := 2\ ,\\
&vc = 2 \wedge n > 0 \wedge (b_0 = 0 \vee \neg n - 1 < N) \longmapsto n := n - 1; vc := 2\ ,\\
&vc = 0 \wedge b_1 > 0 \wedge n + 1 > 0 \longmapsto b_0, n := b_0 - 1, n + 1; vc := 1\ ,\\
&vc = 1 \wedge b_0 > 0 \wedge n - 1 < N \longmapsto b_1, n := b_1 - 1, n - 1; vc := 0\ ,\\
&vc = 1 \wedge b_1 - 1 > 0 \wedge n - 1 > 0 \longmapsto b_1, n := b_1 - 1, n - 1; vc := 1\ ,\\
&vc = 0 \wedge (b_0 - 1 = 0 \vee \neg n + 1 < N) \wedge (b_1 = 0 \vee \neg n + 1 > 0)\\
&\qquad\qquad \longmapsto b_0, n := b_0 - 1, n + 1; vc := 2\ ,\\
&vc = 1 \wedge (b_0 = 0 \vee \neg n - 1 < N) \wedge (b_1 - 1 = 0 \vee \neg n - 1 > 0)\\
&\qquad\qquad \longmapsto b_1, n := b_1 - 1, n - 1; vc := 2
\end{aligned}
$$

Fig. 9. Bounded Buffer transitions (eliminating a producer's guard).

Furthermore, the initial condition generated is:

$$\Theta : vc = 2 \wedge n = 0 \wedge b_0 = 0 \wedge b_1 = 0 \wedge N > 0\ .$$

After a few seconds the method converge in one iteration, returning the same invariant ϕ.

The second test we attempt to eliminate the producer's guard $n < N \wedge b_0 > 0$. This guard release producer that wait in the semaphore. The resulting set of transitions is writed in Fig. 9 and the candidate invariant is

$$
\begin{aligned}
\phi : &(vc = 0 \rightarrow 0 \leq n \wedge n < N \wedge b_0 > 0 \wedge (b_0 = 1 \vee \neg n + 1 < N)) \wedge\\
&(vc = 1 \rightarrow 0 < n \wedge n \leq N \wedge b_1 > 0) \wedge\\
&(vc = 2 \rightarrow (\neg n < N \vee b_0 = 0) \wedge (\neg n > 0 \vee b_1 = 0) \wedge 0 \leq n \wedge n \leq N)\ .
\end{aligned}
$$

After a few seconds the method converge in four iterations, returning the following inductive invariant:

$$\begin{aligned}\varphi : (vc = 0 \rightarrow 0 \leq n \wedge n < N \wedge b_0 > 0 \wedge (n < N - 1 \rightarrow b_0 = 1) \\ \wedge (n > 0 \rightarrow b_0 \leq 2 \vee b_1 \leq 1)) \wedge \\ (vc = 1 \rightarrow 0 < n \wedge n \leq N \wedge b_1 > 0) \wedge (n > 1 \rightarrow b_0 \leq 1 \vee b_1 = 1) \\ \wedge (n < N \rightarrow b_0 \leq 1)) \wedge \\ (vc = 2 \rightarrow (\neg n < N \vee b_0 = 0) \wedge (\neg n > 0 \vee b_1 = 0) \\ \wedge 0 \leq n \wedge n \leq N)\end{aligned}$$

wich is implied by the initial condition Θ. Hence we can eliminate this guard.

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

For the rests of guards ($n > 0 \wedge b_1 > 0$ in producer and $n < N \wedge b_0 > 0$ in consumer) the method detects the unsatisfability of the formula $\Theta \rightarrow \mathcal{B}(\phi, \mathcal{T}, \varphi)$ in the first iteration.

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

Program	Guard	#Trs.	Result	Iterations	Run Time
Producer	Producer	11	eliminated	5	4.16”
Producer	Consumer	10	detects unsat	2	1.23”
Consumer	Producer	10	detects unsat	3	2.60”
Consumer	Consumer	11	eliminated	5	4.17”

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:

$\Theta' : n = 0 \wedge N > 1$ $I : \ 0 \leq n \wedge n \leq N$	
producer:	consumer:
$B_0 : n < N$	$B_1 : n > 1$
$S_0 : n := n + 1$	$S_1 : n := n - 2$

For this example 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 transition system generated have 11 transitions and the point fix calculation converge in 6 iterations at the invariant:

$$\begin{aligned}\varphi : (vc = 0 \rightarrow\ & 0 \leq n \wedge 0 < b_0 \wedge n < N \\ & \wedge (3 \leq n \rightarrow b_1 = 1 \vee \neg 0 < b_1) \\ & \wedge (2 + n < N \wedge 1 \leq n \rightarrow (0 < b_1 \rightarrow (0 < -2 + b_0 \rightarrow b_1 = 1))) \\ & \wedge (3 + n < N \rightarrow (0 < b_1 \rightarrow b_1 = 1 \vee \neg 0 < -3 + b_0)) \\ & \wedge (n + 1 < N \wedge 2 \leq n \rightarrow (0 < b_1 \rightarrow b_1 = 1 \vee \neg 0 < b_0 - 1))) \wedge \\ (vc = 1 \rightarrow\ & n \leq N \wedge 0 < b_1 \wedge 1 < n \\ & \wedge (3 < n \rightarrow b_1 = 1) \\ & \wedge (n < N \wedge 3 \leq n \rightarrow (0 < b_1 - 1 \rightarrow (b_1 = 2 \vee \neg 0 < -2 + b_0)))) \\ & \wedge (1 + n < N \rightarrow (0 < b_1 - 1 \rightarrow (0 < -3 + b_0 \rightarrow b_1 = 2))) \wedge \\ (vc = 2 \rightarrow\ & n \leq N \wedge 0 \leq n \wedge (b_1 = 0 \vee \neg 1 < n) \wedge (b_0 = 0 \vee \neg n < N))\end{aligned}$$

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

Program	Guard	#Trs.	Result	Iterations	Run Time
Producer	Producer	11	detects unsat	6	12.78”
Producer	Consumer	10	detects unsat	3	16.78”
Consumer	Producer	10	detects unsat	4	12.47”
Consumer	Consumer	11	eliminated	7	26.62”

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:

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

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 unsatisfability of the formula $\Theta \rightarrow \mathcal{B}(\phi, \mathcal{T}, \varphi)$. Results are shown in table 3.

Program	Guard	#Trs.	Result	Iterations	Run Time
Producer	Producer	11	detects unsat	6	19.90”
Producer	Consumer	10	detects unsat	2	2.36”
Consumer	Producer	10	detects unsat	3	4.53”
Consumer	Consumer	11	does not converge	20	22850.91”

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:

$\Theta' : x = 0$ $I : \ x \geq 0$	
P: $B_0 : x > 0$ $S_0 : x := x - 1$	V: $B_1 : \mathit{True}$ $S_1 : x := x + 1$

In this problem only the guard $x > 0 \wedge b_0 > 0$ 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

Program	Guard	#Trs.	Result	Iterations	Run Time
P	P	11	eliminated	5	2.48”
P	V	10	eliminated	3	0.91”
V	P	10	detects unsat	2	0.72”
V	V	11	eliminated	3	0.83”

Table 4. General Semaphore Results.

Readers and Writers

The last example is the classical readers and writers problem. The specification of this problem is:

$\Theta' : w = 0 \wedge r = 0$ $I : \ r \geq 0 \wedge (w = 0 \vee w = 1 \wedge r = 0)$	
Writer (entrance): $B_0 : w = 0 \wedge r = 0$ $S_0 : \ w := w + 1$	Reader (entrance): $B_2 : w = 0$ $S_2 : \ r := r + 1$
Writer (exit): $Pre : w > 0$ $B_1 : \ \ True$ $S_1 : \ \ w := w - 1$	Reader (exit): $Pre : r > 0$ $B_1 : \ \ True$ $S_1 : \ \ r := r - 1$

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 γ_τ (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_3 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 strenghtening process is summarized in table 5.

Program	Guard	#Trs.	Result	Iterations	Run Time
Writer (entrance)	Writer (entrance)	39	eliminated	2	7.80”
Writer (entrance)	Writer (exit)	38	eliminated	1	7.99”
Writer (entrance)	Reader (entrance)	38	eliminated	2	11.87”
Writer (entrance)	Reader (exit)	38	eliminated	1	15.35”
Writer (exit)	Write (entrance)	38	unsat detected	3	7.82”
Writer (exit)	Write (exit)	39	eliminated	3	7.66”
Writer (exit)	Reader (entrance)	38	unsat detected	3	8.88”
Writer (exit)	Reader (exit)	38	eliminated	3	9.75”
Reader (entrance)	Writer (entrance)	38	eliminated	2	12.05”
Reader (entrance)	Writer (exit)	38	eliminated	4	12.38”
Reader (entrance)	Reader (entrance)	39	unsat detected	5	19.22”
Reader (entrance)	Reader (exit)	38	eliminated	4	4.11”
Reader (exit)	Writer (entrance)	38	unsat detected	3	5.09”
Reader (exit)	Writer (exit)	38	eliminated	3	8.90”
Reader (exit)	Reader (entrance)	38	eliminated	4	4.09”
Reader (exit)	Reader (exit)	39	eliminated	3	3.75”

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 theorems provers not only for a posteriori verification of systems, but also for their formal construction and optimization.

A Demostrations 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_m) to (out_i) in the program SCC_i can be eliminated.

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

$$\varphi^{j+1}(\bar{x}, vc) = \phi(\bar{x}, vc) \wedge \bigwedge_{\tau \in \mathcal{T}} vc = src(\tau) \rightarrow \text{wp}\,(\tau, \varphi^j_{tgt(\tau)})$$

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

$$\varphi^{j+1} = \bigwedge_{\tau \in \mathcal{T}} vc = src(\tau) \rightarrow \phi_{src(\tau)} \wedge \text{wp}\,(\tau, \varphi^j_{tgt(\tau)}) \tag{10}$$

The term in the conjunction of this formulae corresponding to the transition is:

$$vc = m \rightarrow \phi_m \wedge (B_i \wedge wp.\, S_i.\, (B_i \wedge b_i > 0) \rightarrow wp.\, S_i.\, \varphi_i^j)$$

We prove that the candidate invariant ϕ_m absorb the right hand side of this equation:

$$\begin{array}{ll} & \phi_m \rightarrow (B_i \wedge wp.\, S_i.\, (B_i \wedge b_i > 0) \rightarrow wp.\, S_i.\, \varphi_i^j) \\ \Leftarrow & \{\ \phi_m \rightarrow \varphi_m \text{ by construction of } \phi \text{ (equation 9)}\ \} \\ & \varphi_m \rightarrow (B_i \wedge wp.\, S_i.\, (B_i \wedge b_i > 0) \rightarrow wp.\, S_i.\, \varphi_i^j) \\ \Leftarrow & \{\ \varphi_m \rightarrow \neg B_i \vee b_i = 0 \text{ by equation 8}\ \} \\ & \neg B_i \vee b_i = 0 \rightarrow (B_i \wedge wp.\, S_i.\, (B_i \wedge b_i > 0) \rightarrow wp.\, S_i.\, \varphi_i^j) \\ \equiv & \{\ wp \text{ is conjunctive, } S_i \text{ does not depend on } b_i\ \} \\ & \neg B_i \vee b_i = 0 \rightarrow (B_i \wedge b_i > 0 \wedge wp.\, S_i.\, B_i \rightarrow wp.\, S_i.\, \varphi_i^j) \\ \equiv & \{\ \text{Predicate calculus}\ \} \\ & \mathit{True} \end{array}$$

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 $B_k \wedge b_k > 0$, the resulting strength of the candidate invariant allow to remove the transitions starting from (in_m) to (out_k) and from (in_i) to (out_k).

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

$$vc = m \rightarrow \phi_m \wedge (B_i \wedge wp.\, S_i.\, (B_k \wedge b_k > 0) \rightarrow wp.\, S_i.\, \varphi_k^j)$$

We prove that the candidate invariant ϕ_m absorb the right hand side of this implication:

$$\begin{array}{ll} & \phi_m \rightarrow (B_i \wedge wp.\, S_i.\, (B_k \wedge b_k > 0) \rightarrow wp.\, S_i.\, \varphi_k^j) \\ \Leftarrow & \{\ \phi_m \rightarrow (B_i \rightarrow wp.\, S_i.\, (\neg B_k \vee b_k = 0))\ \text{by equation 9}\ \} \\ & (B_i \rightarrow wp.\, S_i.\, (\neg B_k \vee b_k = 0)) \rightarrow (B_i \wedge wp.\, S_i.\, (B_k \wedge b_k > 0) \rightarrow \\ & wp.\, S_i.\, \varphi_k^j) \\ \equiv & \{\ \text{Predicate calculus}\ \} \\ & wp.\, S_i.\, (\neg B_k \vee b_k = 0) \wedge B_i \wedge wp.\, S_i.\, (B_k \wedge b_k > 0) \rightarrow wp.\, S_i.\, \varphi_k^j \\ \equiv & \{\ wp \text{ is conjunctive}\ \} \\ & B_i \wedge wp.\, S_i.\, ((\neg B_k \vee b_k = 0) \wedge B_k \wedge b_k > 0) \rightarrow wp.\, S_i.\, \varphi_k^j \\ \equiv & \{\ \text{Predicate calculus}\ \} \\ & B_i \wedge wp.\, S_i.\, \mathit{False} \rightarrow wp.\, S_i.\, \varphi_k^j \\ \equiv & \{\ \text{Law of Exclude Miracle}\ \} \\ & B_i \wedge \mathit{False} \rightarrow wp.\, S_i.\, \varphi_k^j \\ \equiv & \{\ \text{Predicate calculus}\ \} \\ & \mathit{True} \end{array}$$

The proof for the other case is similar. The term in the equation 10 corresponding to the transition is:

$$vc = i \rightarrow \phi_i \wedge (wp.\,(b_i := b_i - 1; S_i).\,(B_k \wedge b_k > 0) \rightarrow wp.\,(b_i := b_i - 1; S_i).\,\varphi_k^j)$$

We prove that the candidate invariant ϕ_i absorb the right hand side of this implication:

$$\begin{array}{ll}
 & \phi_i \rightarrow (wp.\,(b_i := b_i - 1; S_i).\,(B_k \wedge b_k > 0) \rightarrow wp.\,(b_i := b_i - 1; S_i).\,\varphi_k^j) \\
\Leftarrow & \{\ \phi_i \rightarrow wp.\,(b_i := b_i - 1; S_i).\,(\neg B_k \vee b_k = 0) \text{ by equation 9 }\} \\
 & wp.\,(b_i := b_i - 1; S_i).\,(\neg B_k \vee b_k = 0) \rightarrow (wp.\,(b_i := b_i - 1; S_i).\,(B_k \wedge b_k > \\
 & 0) \rightarrow wp.\,(b_i := b_i - 1; S_i).\,\varphi_k^j) \\
\equiv & \{\ \text{Predicate calculus }\} \\
 & wp.\,(b_i := b_i - 1; S_i).\,(\neg B_k \vee b_k = 0) \wedge wp.\,(b_i := b_i - 1; S_i).\,(B_k \wedge b_k > \\
 & 0) \rightarrow wp.\,(b_i := b_i - 1; S_i).\,\varphi_k^j \\
\equiv & \{\ wp \text{ is conjunctive }\} \\
 & wp.\,(b_i := b_i - 1; S_i).\,((\neg B_k \vee b_k = 0) \wedge B_k \wedge b_k > 0) \rightarrow wp.\,(b_i := \\
 & b_i - 1; S_i).\,\varphi_k^j \\
\equiv & \{\ \text{Predicate calculus }\} \\
 & wp.\,(b_i := b_i - 1; S_i).\,\mathit{False} \rightarrow wp.\,(b_i := b_i - 1; S_i).\,\varphi_k^j \\
\equiv & \{\ \text{Law of Exclude Miracle }\} \\
 & \mathit{False} \rightarrow wp.\,(b_i := b_i - 1; S_i).\,\varphi_k^j \\
\equiv & \{\ \text{Predicate calculus }\} \\
 & \mathit{True}
\end{array}$$

A.3

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

$$\begin{array}{ll}
 & \varphi_{SBD} \rightarrow F_m \\
\Leftarrow & \{\ \varphi_{SBD} \rightarrow (vc = m \rightarrow \neg B_i \vee b_i = 0) \text{ by equation 8, } F_m \text{ definition} \\
 & \text{(equation 9) }\} \\
 & (vc = m \rightarrow \neg B_i \vee b_i = 0) \rightarrow (vc = m \wedge B_i \ \rightarrow\ wp.\,S_i.\,(\neg B_i \vee b_i = 0)) \\
\equiv & \{\ \text{Predicate calculus }\} \\
 & vc = m \wedge b_i = 0 \wedge B_i \ \rightarrow\ wp.\,S_i.\,(\neg B_i \vee b_i = 0) \\
\equiv & \{\ wp \text{ is disjunctive because } S_i \text{ is deterministic }\} \\
 & vc = m \wedge b_i = 0 \wedge B_i \ \rightarrow\ wp.\,S_i.\,(\neg B_i) \vee wp.\,S_i.\,b_i = 0 \\
\equiv & \{\ S_i \text{ does not depend on } b_i \ \} \\
 & vc = m \wedge b_i = 0 \wedge B_i \ \rightarrow\ wp.\,S_i.\,(\neg B_i) \vee b_i = 0 \\
\equiv & \{\ \text{Predicate calculus }\} \\
 & \mathit{True}
\end{array}$$

A.4

We can also simplify the transitions going from (in_m) to (out_m):

$$vc = m \wedge B_i \wedge wp.\, S_i.\, \Big(\bigwedge_{0 \le j < m} \neg B_j \vee b_j = 0\Big) \;\longmapsto\; S_i\,; vc := m\ ,$$

with $i \in \{0, \cdots, m-1\}$.

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

$$vc = m \to \phi_m \wedge \Big(B_i \wedge wp.\, S_i.\, \Big(\bigwedge_{0 \le j < m} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big)$$

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

$$\begin{array}{ll}
 & \phi_m \wedge \Big(B_i \wedge wp.\, S_i.\, \Big(\bigwedge_{0 \le j < m} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big) \\
\equiv & \{\ wp \text{ is conjunctive }\ \} \\
 & \phi_m \wedge \\
 & \Big(B_i \wedge wp.\, S_i.\, (\neg B_i \vee b_i = 0) \wedge wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big) \\
\equiv & \{\ \text{Predicate calculus }\ \} \\
 & \phi_m \wedge \Big(\neg B_i \vee \neg wp.\, S_i.\, (\neg B_i \vee b_i = 0) \vee \Big(wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big)\Big) \\
\equiv & \{\ wp \text{ is disjunctive because } S_i \text{ is deterministic }\ \} \\
 & \phi_m \wedge \Big(\neg B_i \vee \neg (wp.\, S_i.\, \neg B_i \vee b_i = 0) \vee \Big(wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big)\Big) \\
\equiv & \{\ \phi_m \to (\neg B_i \vee b_i = 0) \text{ by equations 8 and 9 }\ \} \\
 & \phi_m \wedge (\neg B_i \vee b_i = 0) \wedge \\
 & \Big(\neg B_i \vee \neg (wp.\, S_i.\, \neg B_i \vee b_i = 0) \vee \Big(wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big)\Big) \\
\equiv & \{\ \text{Predicate calculus }\ \} \\
 & \phi_m \wedge (\neg B_i \vee b_i = 0) \wedge \\
 & \Big(\neg B_i \vee \Big(wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big)\Big) \\
\equiv & \{\ \phi_m \to (\neg B_i \vee b_i = 0) \text{ by equations 8 and 9 }\ \} \\
 & \phi_m \wedge \Big(\neg B_i \vee \Big(wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big)\Big) \\
\equiv & \{\ \text{Predicate calculus }\ \} \\
 & \phi_m \wedge \Big(B_i \wedge wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \to wp.\, S_i.\, \varphi_m^j\Big)
\end{array}$$

Hence, the simplified transitions are:

$$vc = m \wedge B_i \wedge wp.\, S_i.\, \Big(\bigwedge_{\substack{0 \le j < m \\ i \ne j}} \neg B_j \vee b_j = 0\Big) \;\longmapsto\; S_i\,; vc := m\ ,$$

with $i \in \{0, \cdots, m-1\}$.

References

[1] G. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Reading, Massachusetts, USA, 1999.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In R. Alur and D. A. Peled, editors, *Proceedings of the* 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[3] N. Bjorner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theor. Comput. Sci.*, 173(1):49–87, 1997.

[4] E. W. Dijkstra. A tutorial on the split binary semaphore. `http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD703.PDF`, Mar. 1979.

[5] E. W. Dijkstra and C. S. Scholten. *Predicate calculus and program semantics*. Springer-Verlag New York, Inc., 1990.

[6] R. Hoogerwoord. Two applications of the split binary semaphore. Course notes RH221, 1990.

[7] J. L. W. Kessels. An alternative to event queues for synchronization in monitors. *Commun. ACM*, 20(7):500–503, 1977.

[8] Z. Manna and A. Pnueli. On the faithfulness of formal models. In *Mathematical Foundations of Computer Science*, pages 28–42, 1991.

[9] A. Martin and J. van de Snepscheut. Design of synchronization algorithms. *Constructive Methods in Computing Science*, pages 445–478, 1989.

[10] L. C. Paulson. The Isabelle reference manual. `http://isabelle.in.tum.de/doc/ref.pdf`, 2004.

[11] F. B. Schneider. *On Concurrent Programming*. Graduate texts in computer science. Springer-Verlag New York, Inc., 1997.

[12] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 113–127, Genova, Italy, Apr. 2001. Springer-Verlag.