Generación de invariantes para implementar eficientemente Regiones Críticas Condicionales

Damián Barsotti
Javier O. Blanco
Fa.M.A.F., Universidad Nacional de Córdoba
Ciudad Universitaria, 5000 Cordoba, Argentina
{damian,blanco}@famaf.unc.edu.ar

25 de julio de 2007

Resumen

La técnica de semáforos binarios divididos (SBS) puede ser usada para implementar regiones críticas condicionales. Dada una especificación de un problema de esta clase, SBS brinda tanto los programas que lo implementan como los invariantes que aseguran su corrección. Aplicando la técnica a casos particulares se encuentran programas que admiten simplificaciones.

Este trabajo se concentra en el desarrollo de un procedimiento automático para obtener estas simplificaciones. El procedimiento consiste en hacer una búsqueda de nuevos invariantes que avalen la corrección de las simplificaciones. Para esto usamos técnicas de generación de invariantes, en particular propagación hacia atrás de las precondiciones más débiles. Su implementación fue realizada utilizando los demostradores Isabelle/HOL y CVC Lite para las demostraciones de validez y simplificaciones de las fórmulas lógicas envueltas en el proceso. El método fue probado sobre diferentes ejemplos clásicos de programación concurrente.

Abstract

Split binary semaphores (SBS) can be used to implement conditional critical regions. Given a specification of a program which use these constructs, the SBS technique produces not only the programs which implement the regions but the invariants which ensure their correctness as well. In most examples the programs obtained are suitable for further simplifications.

This work focus on the development of an automatic procedure to obtain these simplifications. We search for new (stronger) invariants which ensure correctness of the simplifications. For this aim, we use automatic invariant generation, in particular back-propagation techniques for weakest preconditions. Its implementation was developed using the Isabelle/HOL and CVC Lite provers for checking correctness and simplify logical formulae involved in the procedure. This method is illustrated on many classical concurrent programming examples.

Keywords: Programación Concurrente, Theorem Prover, SMT Solver, Semáforos Binarios Divididos, Invariantes, Lenguajes de Programación.

1. Introducción

Una gran cantidad de problemas concurrentes pueden ser resueltos usando los conceptos de espera condicional y exclusión mutua. Por ejemplo en [2] se presenta un método relativamente general para resolver problemas de sincronización usando el *await condicional* $\langle \textbf{await } B \rightarrow S \rangle$ el cual realiza ambas ideas (espera condicional y exclusión mutua). Esta construcción, sin embargo, no se espera que sea parte de la implementación final del programa concurrente, dado que implementar este tipo de instrucción *de manera general* es prohibitivamente ineficiente.

Diversos mecanismos de sincronización se han propuesto los cuales son menos abstractos que el await condicional y que pueden usarse para implementar esta construcción. Esta perdida de abstracción introduce nuevas necesidades de prueba de los programas, incrementando la complejidad de estos y por consiguiente generando muchas nuevas posibilidades de cometer errores. La presencia de errores sutiles, tanto de corrección (safety) como de progreso (en particular la posibilidad de deadlocks o livelocks) ha sido desafortunadamente más la regla que la excepción. Toda propuesta de nuevas primitivas de sincronización ha sido siempre una solución de compromiso entre facilidad de uso y posibilidad de reducir las penalidades en eficiencia.

Las *regiones críticas condicionales* (ver por ej. [1, 16]) son la construcción más cercana a los await condicionales y por lo tanto las más fáciles de usar correctamente. La única diferencia con los await condicionales es que es necesario declarar las variables compartidas sobre las cuales se operará en exclusión mutua y que podrán aparecer en las condiciones. Una construcción bastante similar en su nivel de abstracción a ésta es el *monitor con señalamiento automático* propuesto por Brinch Hansen y Hoare en [11, 12]. En este tipo de monitores el wait no se realiza sobre una variable de condición sino directamente sobre una expresión booleana. Las operaciones de signal son directamente eliminadas por lo cual la condición debe ser reevaluada cada vez que algún proceso del monitor salga de una región crítica. Hoare sugiere que usar estos monitores puede ser mucho más seguro pero presenta dudas respecto de la posible ineficiencia de reevaluar siempre la guarda. Comparando las reglas de demostración de estos monitores respecto de los distintos tipos de monitores con señalamiento explícito [6] puede corroborarse el argumento de Hoare. El trabajo de Kessels [14] plantea una manera de implementar los monitores con señalamiento automático de una manera relativamente eficiente siempre y cuando las condiciones no dependan de los parámetros de las llamadas al monitor (es decir, que sean una generalización modesta de las regiones críticas condicionales). Esta implementación es en esencia similar a la del uso de *semáforos binarios divididos* (SBS, del inglés split binary semaphores) que describiremos aquí. No sólo Kessels da esta implementación sino que sugiere también que puede hacerse aún más eficiente aplicando técnicas para eliminar chequeos innecesarios en los puntos en los cuales puede deducirse formalmente que una condición será falsa. Dado que las secciones críticas suelen ser pequeñas pero son invocadas numerosas veces, estos pequeños ahorros pueden representar incrementos drásticos en la eficiencia de los programas. El desarrollo en los últimos años de los demostradores (semi)automáticos de teoremas brinda el contexto en el cual una parte interesante de estas simplificaciones puede hacerse de manera (semi)automática, abriendo las puertas a reducir significativamente las penalidades en eficiencia por el uso de construcciones de muy alto nivel para la construcción de programas concurrentes.

Los SBS permiten implementar de manera eficiente las regiones críticas condicionales [9, 16]. La técnica SBS brinda tanto los programas que implementan las secciones críticas condicionales como los invariantes iniciales que aseguran su corrección. Aplicando la técnica a casos particulares se encuentran programas que admiten simplificaciones.

Este trabajo se concentra en el desarrollo de un procedimiento para verificar la corrección

de estas simplificaciones de forma automática. Nos enfocamos particularmente la eliminación de guardas en las sentencias condicionales finales de los programas, pero el procedimiento encontrado permite su generalización a otros tipos de simplificaciones.

El procedimiento consiste en hacer una búsqueda de nuevos invariantes que avalen la corrección de las simplificaciones. Su implementación así como una breve reseña de las herramientas usadas se hará en la sección 4.

Para hacer estas eliminaciones modelamos los programas generados por medio de sistemas de transiciones guardados. Con ello podemos aplicar al problema técnicas convencionales de generación automática de invariantes. La técnica que utilizamos es *propagación hacia atrás*[5]. Esta técnica permite constatar la invariancia de aserciones sobre los sistemas de transición. La ventaja que posee la misma sobre otras técnicas (como propagación hacia adelante) es que los invariantes obtenidos son fórmulas libres de cuantificadores. Con esto, la obtención de los mismos puede hacerse de forma mecánica con la ayuda de demostradores automáticos de teoremas para fórmulas de primer orden sin cuantificadores (o chequeadores de validez). En particular usaremos CVC Lite [3]. Este chequeador provee varias teorías interpretadas, incluyendo aritmética lineal, vectores, funciones, etc., lo cual es apropiado para la obtención de invariantes.

Las técnicas de propagación están basadas en el cálculo de un punto fijo para un transformador de fórmulas. Una de las ventajas de la propagación hacia atrás es que la secuencia de aproximaciones es usualmente finita. Desafortunadamente las fórmulas producidas en el proceso son generalmente grandes. Usaremos técnicas de simplificación implementadas en CVC Lite y en Isabelle/HOL [13] para aliviar este problema.

El artículo está estructurado como sigue. En la sección 2 describiremos la técnica SBS. En la sección 3 presentaremos las consideraciones teóricas en las cuales se basa el trabajo. En la sección 4 se describe la solución al problema en cuestión. Los resultados de aplicar esta técnica a diversos ejemplos se presentan en la sección 5. La sección 6 establece algunas conclusiones y direcciones en las cuales el trabajo puede ser continuado.

2. SBS

Los semáforos binarios pueden asegurar de manera muy simple exclusión mutua y por lo tanto son buenos candidatos para implementar regiones críticas. Una manera particular de usar los semáforos binarios provee un método para implementar regiones críticas condicionales. Describiremos aquí estas ideas brevemente, remitiendo a la literatura para una presentación más completa [1, 16, 9, 15, 4].

Un conjunto $\{s_0, \ldots, s_n\}$ de semáforos binarios se denominará *semáforo binario dividido* (SBS, del inglés *split binary semaphore*) si en cualquier momento de la ejecución del programa a lo sumo uno de ellos toma el valor 1. Esto es equivalente a requerir la invariancia de la siguiente propiedad:

$$0 \leq \langle \sum i : 0 \leq i \leq n : s_i \rangle \leq 1$$

Toda sección crítica comenzará entonces con una operación P sobre alguno de los elementos del SBS y terminará con un V sobre un elemento del mismo conjunto (no necesariamente el mismo). El invariante garantiza entonces exclusión mutua entre estas dos operaciones.

Además de la exclusión mutua, los SBS satisfacen la siguiente *regla del dominó* [15]. Si la ejecución de una sección crítica termina con una operación V sobre un semáforo s, entonces la próxima operación P deberá ocurrir sobre el mismo semáforo s. Esto permite asumir la precondición de cualquier operación V como postcondición de su correspondiente operación P.

Esta regla puede formularse en términos axiomáticos como la invariancia global de la siguiente aserción:

$$\varphi_{SBS} : \langle \forall s :: s = 0 \vee I_s \rangle$$

siempre que la aserción I_s sea válida antes de cada operación V asumiendo que cada I_s vale luego de su correspondiente operación P.

Para implementar regiones críticas condicionales usaremos los SBS de la siguiente manera. Asociaremos un semáforo binario del SBS a cada condición diferente. Será también necesario un semáforo más para el caso en el cual ninguna condición es válida. Luego, toda región crítica estará dinámicamente prefijada por un P asociado con su precondición. Debe tenerse cierto cuidado para introducir suficientes operaciones V para asegurar progreso. Ilustraremos el método con dos regiones condicionales para facilitar la notación. La extensión a un número arbitrario es inmediata y el resultado puede verse en la sección 4 (figura 3).

Se quieren ejecutar atómicamente las sentencias $\mathrm{S_o}$, $\mathrm{S_1}$ asumiendo como precondiciones respectivamente B_0 y B_1. Supongamos además que las regiones críticas deben preservar cierto invariante I. Usaremos entonces un SBS compuesto por dos semáforos s_0, s_1 uno para cada condición y otro m para cuando ninguna de las dos valga. Dos contadores b_0, b_1 serán necesarios para contar la cantidad de procesos comprometidos respectivamente con la ejecución de $\mathrm{P}.s_0, \mathrm{P}.s_1$ y poder así asegurar la ausencia de deadlocks.

El siguiente invariante caracteriza la solución basada en SBS

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

La figura 1 muestra el programa obtenido por la técnica SBS, junto con su anotación completa, para la primer región crítica condicional.

$\underline{\mathrm{SCC_o}}$

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

Figura 1: Componente generada por la técnica SBS

3. Verificación de Invariantes

Las siguientes definiciones y conceptos están basados en los trabajos [17, 5]. Sea Σ un lenguaje de primer orden con símbolos interpretados en dominios concretos como booleanos, enteros, reales, etc. Sea $\mathcal{F}$ el conjunto de fórmulas de primer orden sobre Σ con variables libres contenidas en un conjunto finito de símbolos de variables $\mathcal{V}$ (tipadas). Usaremos como abreviatura de la secuencia de variables $x_0, \cdots, x_n$ a $\bar{x}$. De la misma forma, si un término del lenguaje Σ (posiblemente una formula en $\mathcal{F}$) posee variables libres en $\{x_0, \cdots, x_n\} \subseteq \mathcal{V}$, lo escribiremos como $t(\bar{x})$.

Un *sistema de transiciones guardado* es una tupla $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{L}, \mathcal{T} \rangle$, donde $\mathcal{V} = \{x_0, \cdots, x_n, vc\}$ (vc se denomina variable de control o "program counter"), $\Theta \in \mathcal{F}$ es la condición inicial del sistema, $\mathcal{L}$ es un conjunto finito de valores denominados locaciones y $\mathcal{T}$ es un conjunto finito de transiciones. Cada $\tau \in \mathcal{T}$ puede especificarse como

$$vc = l_\tau \wedge \gamma_\tau(\bar{x}) \;\longmapsto\; \bar{x} := \bar{e}_\tau(\bar{x})\,; vc := l'_\tau \tag{1}$$

con $l_\tau \in \mathcal{L}$ el origen de la transición, $l'_\tau \in \mathcal{L}$ el destino de la misma, $\gamma_\tau(\bar{x})$ una fórmula con variables libres contenidas en $\mathcal{V}/\{vc\}$ denominada *guarda de la transición* y $\bar{e}_\tau(\bar{x})$ es una secuencia de términos de Σ con variables libres contenidas en $\mathcal{V}/\{vc\}$ de la misma longitud que $\bar{x}$. Para cada transición, la fórmula $\gamma_\tau(\bar{x})$ denota la condición necesaria para la ejecución de la misma y $\bar{x} := \bar{e}_\tau(\bar{x})$ es una asignación múltiple que denota la transformación del estado producida por dicha ejecución.

Dado un sistema $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{L}, \mathcal{T} \rangle$ definimos como la *precondición mas débil de una transición*[1] al transformador de fórmulas

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

con $\tau \in \mathcal{T}$ y φ una fórmula con variables libres contenidas en $\mathcal{V}/\{vc\}$.

Como el conjunto de locaciones $\mathcal{L}$ es finito, una fórmula $\phi \in \mathcal{F}$ puede escribirse como

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

Para un sistema $\mathcal{S}$, la fórmula $\phi(\bar{x}, vc)$ denotará un conjunto de estados para cada locación en $\mathcal{L}$ y cada formula $\phi(\bar{x}, l)$ denotará el conjunto para la locación l en particular. Escribiremos la fórmula $\phi(\bar{x}, l)$ como $\phi_l(\bar{x})$. Utilizando esta notación definiremos el siguiente transformador de fórmulas:

$$\mathrm{WP}\,(\mathcal{T}, \phi)(\bar{x}, vc) = \bigwedge_{l \in \mathcal{L}} vc = l \;\rightarrow \bigwedge_{\substack{\tau \in \mathcal{T} \wedge \\ l_\tau = l}} \mathrm{wp}\,(\tau, \phi_{l'_\tau})(\bar{x}) \;\; . \tag{3}$$

Este transformador define la *precondición más débil de un sistema de transiciones.*

Sea $\mathfrak{R}$ una teoría sobre el lenguaje Σ. Dado un sistema $\mathcal{S} = \langle \mathcal{V}, \Theta, \mathcal{L}, \mathcal{T} \rangle$, una fórmula $\varphi \in \mathcal{F}$ es un *invariante inductivo* del sistema si $\mathfrak{R} \models \Theta \rightarrow \varphi$ y $\mathfrak{R} \models \varphi \rightarrow \mathrm{WP}\,(\mathcal{T}, \varphi)$. Ya que la teoría $\mathfrak{R}$ es fija, no la mencionaremos explícitamente cuando hablemos de validez y satisfabilidad en $\mathfrak{R}$. Una fórmula $\phi \in \mathcal{F}$ será un *invariante* (a secas) si existe un invariante inductivo φ tal que $\models \varphi \rightarrow \phi$.

Para un transformador de fórmulas monótono $\Gamma : \mathcal{F} \mapsto \mathcal{F}$, escribiremos el mayor punto fijo como $\nu X.\Gamma(X)$. Su significado será el usual [10, 5].

[1] Esta es la definición usual de "weakest precondition" de la asignación $\bar{x} := \bar{e}_\tau(\bar{x})$ si se cumple γ_τ.

Dada una fórmula $\phi \in \mathcal{F}$ definiremos el transformador de fórmulas monótono

$$\mathcal{B}(Y) \triangleq \phi \wedge \mathrm{WP}\,(\mathcal{T}, Y)\ . \tag{4}$$

Sea $\varphi_{\mathcal{B}} = \nu X.\mathcal{B}(X)$ el mayor punto fijo de este transformador en un sistema $\mathcal{S}$ dado. Entonces $\varphi_{\mathcal{B}}$ es el predicado más débil tal que $\models \varphi_{\mathcal{B}} \rightarrow \phi$ y $\models \varphi_{\mathcal{B}} \rightarrow \mathrm{WP}\,(\mathcal{T}, \varphi_{\mathcal{B}})$. Por lo tanto, si $\models \Theta \rightarrow \varphi_{\mathcal{B}}$ entonces ϕ es un invariante y $\varphi_{\mathcal{B}}$ es un invariante inductivo del sistema.

Al ser $\mathcal{B}$ un transformador monótono, si la secuencia $\varphi^{(0)}, \varphi^{(1)}, \cdots, \varphi^{(i+1)}, \cdots$

$$\underbrace{\mathit{True}}_{\varphi^{(0)}} \leftarrow \underbrace{\mathcal{B}(\varphi^{(0)})}_{\varphi^{(1)}} \leftarrow \cdots \leftarrow \underbrace{\mathcal{B}(\varphi^{(i)})}_{\varphi^{(i+1)}} \cdots \tag{5}$$

converge en una cantidad finita de pasos (comenzando de *True*), entonces su límite es $\varphi_{\mathcal{B}}$. Con esta propiedad es posible explorar el espacio abstracto de estados utilizando la técnica de *propagación hacia atrás* [5]: dada una fórmula ϕ (denominada invariante candidato) obtendremos el punto fijo $\varphi_{\mathcal{B}}$ si la secuencia converge en una cantidad finita de pasos. Entonces, para verificar que ϕ es un invariante del sistema, tendremos que demostrar $\models \Theta \rightarrow \varphi_{\mathcal{B}}$. Además, si algún $\varphi^{(i)}$ de la secuencia calculada verifica $\not\models \Theta \rightarrow \varphi^{(i)}$ (la fórmula $\Theta \rightarrow \varphi^{(i)}$ es no satisfacible) entonces, por monoticidad de la secuencia, el punto fijo $\varphi_{\mathcal{B}}$ también lo verificará. Si esto sucede podemos concluir que la fórmula ϕ no es un invariante del sistema.

4. Proceso de Eliminación de Guardas

Las implementaciones de regiones críticas condicionales que brinda la técnica SBS permiten en general ciertas optimizaciones de los programas resultantes. En la mayor parte de los programas generados para distintos problemas se verifica que algunas condiciones en el comando guardado final nunca serán satisfechas al ser evaluadas en todos los posibles estados de ejecución [9, 16]. Este trabajo se concentra en la eliminación de estas guardas superfluas dentro de los programas generados.

En general, para decidir que condiciones pueden ser eliminadas, modelamos la ejecución de los programas generados con sistemas de transiciones guardados. Resumiendo, el proceso de eliminación de guardas consiste de las siguientes etapas:

1. A partir de una especificación de un problema particular de regiones críticas condicionales generamos un sistema de transiciones que modela la ejecución de los programas obtenidos mediante la técnica SBS, y un invariante inductivo sobre el mismo que asegura la exclusión mutua condicional.

2. Eligiendo una condición (del comando guardado final) y fortaleciendo el invariante obtenemos un invariante candidato que captura, no solo la exclusión mutua condicional, sino también la imposibilidad de ejecución de la condición a eliminar.

3. Finalmente, utilizando la técnica de propagación hacia atrás, verificamos si el invariante candidato es un invariante del sistema. Con ello podemos decidir si la guarda del programa se puede eliminar.

Las etapas 1 y 2 fueron implementadas en el módulo *Generador de Transiciones* escrito en lenguaje ML. La etapa restante fue implementada en el modulo *Verificador de Invariantes* escrito en el mismo lenguaje y utilizando como proceso externo el probador de teoremas

CVC Lite [3]. Para hacer más eficiente el cálculo del punto fijo (ecuación 5 en página anterior) realizado por este módulo, incluimos técnicas de simplificación del sistema transiciones y de las formulas obtenidas en cada paso de este cálculo. Las primeras fueron implementadas en ML (dentro del primer módulo) y las segundas utilizando CVC Lite y el probador de teoremas Isabelle [13] junto con algunas técnicas de simplificación propias codificadas en ML (dentro del segundo módulo). La figura 2 muestra un diagrama esquemático de todo el proceso.

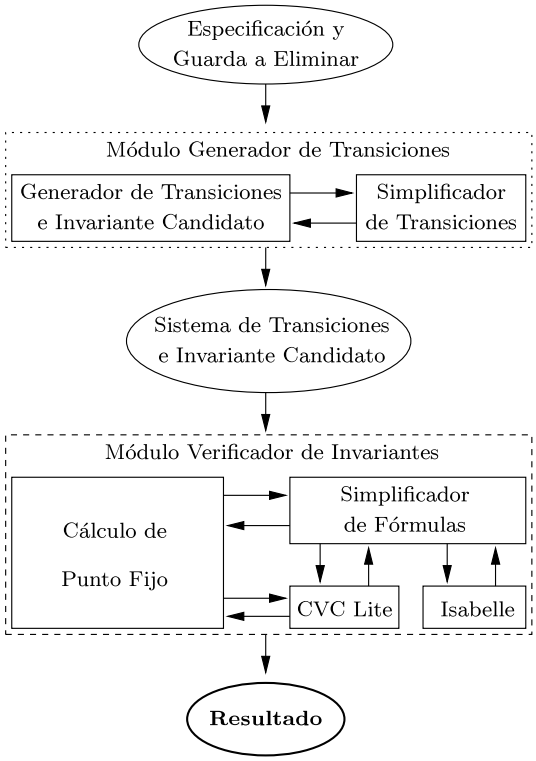

Figura 2: Diagrama del Proceso.

A continuación describiremos los detalles de implementación de ambos módulos.

Generador de Transiciones Este módulo genera el sistema de transiciones y el invariante candidato a partir de una especificación de un problema de regiones críticas condicionales y la información del comando guardado final que se desea eliminar. En su forma más general, un problema de regiones críticas condicionales puede ser especificado como:

- una secuencia de m programas $S_0, \cdots, S_{m-1}$ a ejecutarse en exclusión mutua condicional,
- m condiciones $B_0, \cdots, B_{m-1}$ de ejecución para los programas anteriores respectivamente,
- un invariante global del sistema I que debe ser mantenido por las ejecuciones de dichos programas[2].

Para simplificar la implementación del generador de invariantes, vamos a pedir que los programas $S_0, \cdots, S_{m-1}$ sean simples asignaciones múltiples.

[2]Las condiciones y el invariante global son redundantes, en el sentido que unas pueden ser obtenidos del otro de forma mecánica o viceversa. Esta previsto agregar esta funcionalidad en versiones futuras del software.

A partir de estos datos iniciales, la técnica SBS brinda m nuevos programas $SCC_0, \cdots$, SCC_{m-1} tal que, si son ejecutados de forma concurrente, verificarán la propiedad de exclusión mutua y el invariante global. La figura 3 muestra el esquema de un programa SCC_i generado por SBS.

$\underline{SCC_i}$

P.s_m ;
if $B_i \rightarrow$ skip
$\square\ \neg B_i \rightarrow b_i := b_i + 1$;
V.s_m ;
P.s_i ;
$b_i := b_i - 1$
fi;
S_i ;
if $B_0 \wedge b_0 > 0 \rightarrow$ V.s_0
$\vdots$ $\vdots$
$\square\ B_{m-1} \wedge b_{m-1} > 0 \rightarrow$ V.s_{m-1}
$\square\ \bigwedge_{0 \leq j < m} \neg B_j \vee b_j = 0 \rightarrow$ V.s_m
fi

Figura 3: Programa generado por SBS.

Analizando estos programas puede observarse que cualquier proceso comienza una traza de ejecución con una operación P y la finaliza en una operación V. Esto es una característica de la técnica SBS: semáforos binarios divididos asegura exclusión mutua entre pares de sentencia P y V, esto es, como máximo un semáforo está activado en cualquier punto de ejecución [9]. Además, por la *regla del dominó* [15] podemos asegurar que dinámicamente cada operación V será seguida por su correspondiente operación P (ejecutada por otro proceso) aplicada sobre el mismo semáforo. En base a estos hechos, modelamos cada traza de ejecución posible sobre cada programa SCC_i como una transición del sistema a generar (con la forma de la ecuación 1) y cada locación identificará que semáforo está activo ($\mathcal{L} = \{s_0, \cdots, s_m\}$).

Para obtener las guardas γ_τ de cada transición, encontramos la condición que se debe satisfacer al ejecutar la traza correspondiente. Estas condiciones vienen dadas por las guardas en los comandos guardados inicial y final que se evalúan al recorrer las trazas. Como, la interpretación de transiciones requiere que la guarda de cada una sea evaluada con anterioridad, para obtenerlas propagaremos hacia atrás las condiciones de los comandos guardados con el clásico transformador *weakest precondition* [10]. Por ejemplo, en el anterior programa SCC_i, la transición desde P.s_m hasta V.s_0 tendrá la guarda $\gamma_\tau : B_i \wedge \mathrm{wp}\,(S_i, B_0 \wedge b_0 > 0)$.

Las asignaciones de cada transición pueden obtenerse directamente calculando el cambio de estado producido por la traza correspondiente. Por ejemplo, en el anterior programa SCC_i, la transición desde P.s_i hasta V.s_0 tendrá las asignaciones $b_i := b_i - 1; S_i; vc := s_0$. En este caso, la transición total resultante será $vc = s_i \wedge \mathrm{wp}\,(b_i := b_i - 1\,; S_i, B_0 \wedge b_0 > 0) \longmapsto b_i := b_i - 1\,; S_i\,;$ $vc := s_0$.

Existen problemas de exclusión mutua condicional que requieren agregar además en su especificación ciertas *precondiciones de contorno* para algunos programas de la secuencia $S_0, \cdots, S_{m-1}$. Esto sucede por ejemplo en el caso del clásico problema "Lectores/Escritores" [1] donde los programas correspondientes a las salidas de los lectores y escritores tienen como precondición la existencia de al menos un proceso del mismo tipo ejecutándose. En la especificación del problema se pueden agregar estas precondiciones de contorno como $r > 0$ y $w > 0$ respectivamente.

La manera en que incluimos las precondiciones de contorno en el sistema de transición generado es fortaleciendo las guardas de las transiciones de los programas que las contienen: si el programas S_i tiene una precondición de contorno P, tomamos todas las transiciones generadas a partir de este programa y reemplazamos cada guarda γ_τ por $P \wedge \gamma_\tau$ en dichas transiciones.

La generación del invariante candidato se produce en dos pasos: primero se genera el invariante inductivo que asegura la exclusión mutua condicional y después se fortalece el mismo de forma que capture la imposibilidad de ejecución de la guarda a eliminar.

Para implementar el primer paso utilizamos el mismo invariante que brinda la técnica SBS: debido a que asociamos semáforos con locaciones, este invariante sirve perfectamente para el sistema de transiciones generado y además puede demostrarse que es inductivo. Dada una especificación de un problema de regiones críticas condicionales dicha técnica genera el invariante inductivo:

$$\varphi_{SBS} : \bigwedge_{0 \leq i \leq m} vc = s_i \rightarrow \varphi_{s_i} \tag{6}$$

con

$$\varphi_{s_i} : \; B_i \wedge b_i > 0 \wedge I \quad \text{con } 0 \leq i < m \;, \qquad \varphi_{s_m} : \; (\bigwedge_{0 \leq j < m} \neg B_j \vee b_j = 0) \wedge I$$

e I el invariante global de la especificación.

Para obtener el invariante candidato fortalecemos φ_{SBS} con la precondición más débil (usando el transformador wp) de la guarda a eliminar negada, sobre la trazas que llegan a dicha guarda. Con esto, el invariante candidato resultante denotará la imposibilidad de ejecución de las trazas involucradas. Por ejemplo, si queremos eliminar la guarda $\Box B_k \wedge b_k > 0$ (para algún $k \in \{0, \cdots, m-1\}$) en el programa S_i, el invariante candidato ϕ será:

$$\phi : \; \varphi_{SBD} \wedge F_m \wedge F_i$$

con

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

siendo F_m el fortalecimiento correspondiente a la traza que comienza en el comando P.s_m y F_i el correspondiente a la traza que comienza en P.s_i, ambas terminando en la guarda a eliminar.

Además, el módulo efectúa simplificaciones en el conjunto de transiciones resultantes. Por ejemplo, una de estas consiste en la eliminación de la transición envuelta en la guarda a eliminar. Para una descripción detallada de las mismas y sus demostraciones de corrección remitirse a [4].

Cabe aclarar que, en la implementación del módulo, las locaciones se representaron con números enteros ($\mathcal{L} = \{0, \cdots, m\}$) y usando el resultado en la ecuación 2 (página 5) las formulas se almacenaron en arreglos indexados en $\mathcal{L}$: en general una fórmula $\phi(\bar{x}, vc) \in \mathcal{F}$ será representada por $[\phi_0(\bar{x}), \cdots, \phi_m(\bar{x})]$ y para el caso particular del invariante φ_{SBS} su representación será $[\varphi_{s_0}, .., \varphi_{s_m}]$. Este diseño estuvo inspirado en el trabajo [17].

Verificador de Invariantes Este módulo toma el sistema de transiciones generado junto con el invariante candidato más una condición inicial inicial Θ y verifica el invariante.

Dadas las fórmulas ϕ y φ (ambas representadas como arreglos) implementamos el operador $\mathcal{B}$ (ecuación 4) como muestra la figura 4 (parte izquierda).

En cada iteración la función calcula el conjunto $\{\tau \in \mathcal{T} : l_\tau = i\}$ (conjunto de transiciones que parten de i) y el resultado es obtenido utilizando la ecuación 3. La función $\mathfrak{R}$-simplify hace simplificaciones en las fórmulas y siempre devuelve fórmulas equivalentes. Esta función

function $\mathcal{B}(\phi, \mathcal{T}, \varphi)$
  for $i \in \mathcal{L}$ **do**
    $\mathcal{T}_i := \{\tau \in \mathcal{T} : l_\tau = i\}$;
    $\varphi'[i] := \phi[i] \ \wedge \ \bigwedge_{\tau \in \mathcal{T}_i} \mathrm{wp}\,(\tau, \varphi[l'_\tau])$;
    $\varphi'[i] := \mathfrak{R}\text{-simplify}(\varphi'[i])$;
  end for
  return φ' ;

function backPropagation$(\phi, \mathcal{T}, k)$
  $\varphi := [\mathit{True}, \cdots, \mathit{True}]$;
  $i := 0$;
  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 noconverge(φ) ;

Figura 4: Programas del Verificador de Invariantes

está implementada utilizando CVC Lite y técnicas de simplificación propias. Para más detalles referirse a [4].

La función backPropagation (figura 4 parte derecha) realiza el cálculo de punto fijo utilizando el operador $\mathcal{B}$. La misma consta básicamente de un bucle donde se calculan las fórmulas $\varphi^{(i)}$ de la ecuación 5. La variable φ almacena estos valores. En cada paso de iteración, si la fórmula $\Theta \rightarrow \mathcal{B}(\phi, \mathcal{T}, \varphi)$ no es satisfacible (chequeando $\not\models \Theta \rightarrow \mathcal{B}(\phi, \mathcal{T}, \varphi)$ con CVC Lite) la función devuelve el valor unsat$(\mathcal{B}(\phi, \mathcal{T}, \varphi))$ indicando que el candidato ϕ no es invariante (ver sección 3). Si esto no sucede, el programa verifica si se ha llegado al punto fijo chequeando $\models \varphi \rightarrow \mathcal{B}(\phi, \mathcal{T}, \varphi)$ (también con CVC Lite). Si esto sucede se devuelve el valor converge(φ) el cual almacena dicho punto fijo. En otro caso se almacena el resultado del operador $\mathcal{B}$. El programa ejecuta a lo sumo k iteraciones, devolviendo el valor noconverge(φ) en el caso que se alcance este límite, indicando que no se pudo decidir la invariancia de ϕ.

5. Resultados

El procedimiento desarrollado fue aplicado a diversos ejemplos clásicos de la programación concurrente. Los ejemplos considerados fueron Semáforos Generales (implementación de semáforos generales con binarios), Productor/Consumidor (Productor/Consumidor que intercambian elementos por medio de un buffer acotado), Productor/Consumido Goloso (ídem pero consumiendo de a 3 elementos), Productor/Consumido Goloso M (ídem pero consumiendo de a m elementos con m una constante) y Lectores/Escritores. La formulación de estos problemas puede encontrarse en [1, 16, 9, 4]. El software más las especificaciones de entrada para los ejemplos pueden encontrarse en `http://www.cs.famaf.unc.edu.ar/~damian/publications/sbdinv/programs/programs.tgz`.

El procedimiento encontró todas las guardas que se podían eliminar (los resultados son los mismos que los obtenidos en la literatura) y en tiempos breves (algunos segundos) para todos los casos, excepto para la eliminación de una guarda en el problema de Productor/Consumidor consumiendo de a m elementos. Notar que este problema es una generalización de los dos anteriores, en los cuales el método terminó exitosamente. Haciendo el cálculo del punto fijo a mano y con la ayuda del mismo software (el software permite ver las formulas $\varphi^{(i)}$ en cada iteración), se puede deducir que las fórmulas de la ecuación 5 claramente no convergen en un

número finito de pasos.

Los resultados fueron obtenidos utilizando una PC con procesador Athlon XP de 1400 MHz. y 256 MB de memoria RAM. El detalle de los mismos (guardas que se pudieron eliminar, run time, cantidad de iteraciones, etc.) puede encontrarse en [4].

6. Conclusiones y Trabajos Futuros

Las regiones críticas condicionales son un patrón de alto nivel para la programación concurrente. La mayor parte de los lenguajes de programación no provee este tipo de construcciones como primitivas debido a que son difíciles de implementar de forma eficiente. Este trabajo muestra un método para optimizar automáticamente implementaciones de regiones críticas condicionales realizadas mediante la técnica SBS. Dado que muchos problemas concurrentes pueden ser resueltos con estas construcciones, el método puede ser muy útil a la hora de crear compiladores que implementen regiones críticas condicionales.

En este trabajo se probó la eficiencia del procedimiento sobre algunos problemas clásicos de concurrencia. Si bien no se dispone de una caracterización de los problemas para los cuales el algoritmo de propagación converge, este funciona bien para una gran cantidad de ejemplos. En su estado actual, el algoritmo podría ser aplicado para la optimización de programas simplemente limitando el número de pasos usando alguna constante que podría determinarse empíricamente. Si bien esto puede dejar fuera programas convergentes, este fenómeno no necesariamente puede ser resuelto en general, dado que el algoritmo tiene que chequear también implicaciones en la aritmética, lo cual no siempre es decidible.

El algoritmo puede mejorarse usando widenings e interpretación abstracta [17, 5, 7] para los casos en los cuales no converge. Hasta el momento solo se intentó utilizar interpretación abstracta sobre el dominio de los poliedros convexos [8]. Con este método solo se encontraron invariantes más débiles que los obtenidos mediante la técnica SBS (ecuación 6) lo cual no permite realizar nuevas simplificaciones. Quedan pendientes para futuros trabajos el empleo de otros dominios de interpretación abstracta.

Referencias

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

[2] G. R. Andrews. A method for solving synchronization problems. *Sci. Comput. Program.*, 13(1):1–21, 1989.

[3] 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.

[4] D. Barsotti and J. O. Blanco. (Im)proving split binary semaphores. Tecnical Report. Available at `http://www.cs.famaf.unc.edu.ar/~damian/publicaciones/sbdinv/SBDwip_ext.pdf`, 2007.

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

[6] P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, 1995.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

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

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

[11] P. B. Hansen. A programming methodology for operating system design. In *IFIP Congress*, pages 394–397, 1974.

[12] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[13] Isabelle Theorem Prover home page. `http://isabelle.in.tum.de`, 2007.

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

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

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

[17] 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.