Un modelo simple para `crond` en TLA+

Maximiliano Cristiá

Ingeniería de Software 1
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

2021

Resumen Se presenta la especificación de una versión simplificada del sistema de ejecución de tareas programadas, presente en los sistemas operativos tipo UNIX, conocido como `crond`. Además se explica cómo especificar requisitos temporales usando timers.

Índice

module *DK_RealTime*

extends *Naturals*

VARIABLES now

$RTTypeInv \triangleq now \in Nat$

$RTInit \triangleq now = 0$

$Tick \triangleq now' = now + 1$

$RTSpec \triangleq RTInit \land \Box[Tick]_{now} \land WF_{now}(Tick)$

THEOREM $RTSpec \Rightarrow \Box RTTypeInv$

Figura 1: Modelo para el tiempo real.

1. Condiciones temporales usando temporizadores

`crond` es un programa UNIX que ejecuta programas cada cierto intervalo de tiempo. En consecuencia, para poder especificar el comportamiento de `crond` necesitamos poder especificar condiciones temporales en TLA+. Especificaremos condiciones temporales por medio de temporizadores, de manera similar a como hicimos en CSP y a como se hace en Statecharts. A cada condición temporal se le asociará un temporizador. Cuando deba controlarse el paso de un cierto período de tiempo se iniciará el temporizador respectivo y se esperará a que este comunique el paso del tiempo estipulado mediante la señal $Timeout$. Si antes de la llegada de $Timeout$ aparece un evento que hace innecesario seguir esperándola, entonces el temporizador puede detenerse. Esta es una descripción bastante cercana a una implementación razonable como para que un programador pueda implementarla fácilmente.

Comenzamos modelando el tiempo real mediante la variable now que cuenta el tiempo desde cierto momento inicial (por ejemplo desde el primero de enero de 1980). El tiempo real avanza discreta pero indefinidamente cada vez que se "ejecuta" la acción $Tick$.

> El tiempo real avanza una unidad de tiempo $\approx Tick$

El temporizador descripto en la Figura 2 es un módulo con una interfaz que permitirá a otros módulos configurar el tiempo límite, iniciar el conteo del tiempo, detenerlo y recibir la señal $Timeout$. El temporizador "percibe" el paso del tiempo pues extiende el módulo $DK\_RealTime$.

> Se configura el temporizador para que cuente el tiempo hasta t unidades de tiempo después de haber sido iniciado $\approx Set(t)$
>
> Se inicia el temporizador $\approx Start$
>
> Se detiene el temporizador $\approx Stop$
>
> El temporizador alcanza la cota de tiempo estipulada $\approx Timeout$

$RTBound$ es un predicado definido en el módulo $RealTime$ de la biblioteca estándar de TLA+ que se usa para especificar restricciones temporales sobre acciones. La signatura del predicado es $RTBound(A, v, \delta, \epsilon)$ donde A es una acción, v es una o más variables de estado y δ y ϵ son números no negativos. Este predicado estipula que se *puede* dar un paso $\langle A \rangle_v$ solo después de que $\langle A \rangle_v$ haya estado continuamente habilitada por δ unidades de tiempo desde

module *Timer*

extends *Naturals, DK_RealTime, RealTime*

VARIABLES *time, running, limit*

$TTypeInv \triangleq time, limit \in Nat \wedge running \in \{\text{“yes”}, \text{“no”}\}$

$sv \triangleq \langle limit, time, running \rangle$

$av \triangleq \langle limit, time, running, now \rangle$

$TInit \triangleq limit = 0 \wedge running = \text{“no”}$

$Set(l) \triangleq \wedge l > 0$
$\wedge running = \text{“no”}$
$\wedge limit' = l$
$\wedge$ UNCHANGED $\langle time, now, running \rangle$

$Start \triangleq \wedge running = \text{“no”}$
$\wedge limit > 0$
$\wedge time' = now$
$\wedge running' = \text{“yes”}$
$\wedge$ UNCHANGED $\langle now, limit \rangle$

$Timeout \triangleq \wedge running = \text{“yes”}$
$\wedge now - time \geq limit$
$\wedge running' = \text{“no”}$
$\wedge$ UNCHANGED $\langle now, time, limit \rangle$

$Stop \triangleq \wedge running = \text{“yes”}$
$\wedge running' = \text{“no”}$
$\wedge$ UNCHANGED $\langle now, time, limit \rangle$

$TNext \triangleq Start \vee Stop \vee Timeout \vee (\exists\, t \in Nat : Set(t))$

$TSpec \triangleq TInit \wedge \Box[TNext]_{sv} \wedge RTBound(Timeout, av, 0, 1)$

Notar que en $RTBound$ usamos av y no sv porque de haber incluido now en sv hubiéramos prohibido el paso del tiempo.

THEOREM $TSpec \Rightarrow \Box\, TTypeInv$

Figura 2: La especificación de un temporizador que luego usaremos para especificar requisitos temporales.

el último paso $\langle A \rangle_v$ (o desde el inicio de la ejecución), y que se *debe* ejecutar antes de que $\langle A \rangle_v$ haya estado continuamente habilitada por más de ϵ unidades de tiempo desde el último paso (o desde el inicio de la ejecución). En otras palabras y simplificando un poco, $RTBound$ dice que A debe ejecutarse dentro del intervalo $[\delta, \epsilon]$ tomado desde la última vez que se ejecutó A siempre y cuando haya estado habilitada todo ese tiempo. En particular, si δ es cero entonces el sistema tiene ϵ unidades de tiempo para ejecutar la acción desde el momento en que queda habilitada. En este caso el predicado $RTBound$ se usa para establecer una cota superior para el tiempo que puede tomarse el sistema para ejecutar la acción. Usado de esa forma representa una forma extrema de vitalidad. La definición de $RTBound$ y del resto del módulo $RealTime$ puede encontrarse en [1, página 125]. Lamport indica que las cotas δ y ϵ son necesarias porque es imposible medir el tiempo con precisión absoluta.

Entonces, para el caso de $Timeout$, con $RTBound$ estamos especificando que se debe ejecutar antes de que pase una unidad de tiempo desde el momento en que queda habilitada, lo que tal vez sea exigir demasiada precisión.

Como dijimos al comienzo de la sección, de aquí en más especificaremos los requisitos temporales utilizando uno o más temporizadores como el que acabamos de describir y no utilizaremos el módulo $RealTime$ definido por Lamport. En este caso no seguiremos la técnica descripta por Lamport porque consideramos que es demasiado declarativa y que por lo tanto no da suficientes pistas al programador. Si lo que se busca es un modelo más abstracto entonces la técnica de Lamport es mejor que la que nosotros proponemos.

En otro orden, decidimos no incluir $Tick$ en $TNext$ porque nos parece que metodológicamente $Tick$ no es una acción del temporizador; es decir, el tiempo real avanza o no independientemente de la existencia del temporizador. Sin embargo, de esta forma $TSpec$ admite ejecuciones en las que el tiempo real, por ejemplo, retrocede. Pero, por otro lado, $Timeout$ se habilitará únicamente cuando se cumpla la precondición $now - time \geq limit$ sin perjuicio de que now haya tomado valores contrarios a la naturaleza. Precisamente es este hecho el que nos lleva a proponer este modelo: el tiempo real no retrocede, sólo avanza, por lo tanto si el temporizador es "guiado" por un dispositivo que cuenta el tiempo real, entonces $Timeout$ funcionará correctamente. Finalmente, dado que quienes programan el temporizador no pueden modificar el valor de now, cualquier implementación correcta funcionará apropiadamente pues deberá permitir que únicamente el entorno, a través del dispositivo físico que cuenta el tiempo, modifique la variable en cuestión.

Si no se tienen en cuenta las consideraciones previas, otra posibilidad para definir $TSpec$ podría haber sido la siguiente:

$$TSpec \triangleq TInit \wedge \Box[TNext]_{sv} \wedge RTBound(Timeout, av, 0, 1) \wedge RTSpec$$

Que puede re-escribirse de la siguiente forma

$$\begin{aligned} TSpec \triangleq\ & TInit \wedge RTInit \\ & \wedge \Box[TNext]_{sv} \wedge \Box[Tick]_{now} \\ & \wedge RTBound(Timeout, av, 0, 1) \wedge WF_{now}(Tick) \\ & \equiv TInit \wedge RTInit \\ & \wedge \Box[TNext \vee Tick]_{av} \\ & \wedge RTBound(Timeout, av, 0, 1) \wedge WF_{av}(Tick) \end{aligned}$$

module *Timers*

extends *Naturals, DK_RealTime, RealTime*

VARIABLES *timers*

$$TTypeInv \triangleq timers \in [Nat \to [t, l : Nat, r : \{\text{“yes”}, \text{“no”}\}]]$$

$$TInit \triangleq timers = [n \in Nat \mapsto [t \mapsto now, l \mapsto 0, r \mapsto \text{“no”}]]$$

$$\begin{aligned} Set(i,lim) \triangleq\ & \land\ lim > 0 \\ & \land\ timers[i].r = \text{“no”} \\ & \land\ timers' = [timers \text{ EXCEPT } ![i] = [t \mapsto @.t, l \mapsto lim, r \mapsto @.r]] \\ & \land\ \text{UNCHANGED } \langle now \rangle \end{aligned}$$

$$\begin{aligned} Start(i) \triangleq\ & \land\ timers[i].r = \text{“no”} \\ & \land\ timers[i].l > 0 \\ & \land\ timers' = [timers \text{ EXCEPT } ![i] = [t \mapsto now, l \mapsto @.l, r \mapsto \text{“yes”}]] \\ & \land\ \text{UNCHANGED } \langle now \rangle \end{aligned}$$

$$\begin{aligned} Timeout(i) \triangleq\ & \land\ timers[i].r = \text{“yes”} \\ & \land\ now - timers[i].t \geq timers[i].l \\ & \land\ timers' = [timers \text{ EXCEPT } ![i] = [t \mapsto @.t, l \mapsto @.l, r \mapsto \text{“no”}]] \\ & \land\ \text{UNCHANGED } \langle now \rangle \end{aligned}$$

$$\begin{aligned} Stop \triangleq\ & \land\ timers[i].r = \text{“yes”} \\ & \land\ timers' = [timers \text{ EXCEPT } ![i] = [t \mapsto @.t, l \mapsto @.l, r \mapsto \text{“no”}]] \\ & \land\ \text{UNCHANGED } \langle now \rangle \end{aligned}$$

$$TNext \triangleq \exists i \in Nat : Start(i) \lor Stop(i) \lor Timeout(i) \lor (\exists t \in Nat : Set(i,t))$$

$$TSpec \triangleq TInit \land \Box[TNext]_{timers} \land (\forall i \in Nat : RTBound(Timeout(i), \langle timers, now \rangle, 0, 1))$$

THEOREM $TSpec \Rightarrow \Box TTypeInv$

Figura 3: La especificación de un arreglo de temporizadores.

1.1. Un arreglo de temporizadores

Para especificar algunos sistemas o propiedades necesitaremos más de un temporizador a la vez. Por lo tanto, en la Figura 3 especificamos un módulo TLA+ desde el cual se pueden manejar varios temporizadores simultaneamente.

2. Requerimientos informales y designaciones para `crond`

En esta versión del sistema `cron` cualquier usuario del sistema puede solicitar la ejecución programada de un programa, y cualquiera puede, sabiendo el número de tarea, eliminarla del sistema. Al solicitar una ejecución programada el usuario debe indicar el programa, el tiempo (en alguna unidad de tiempo), contado desde el momento de efectuar la petición, cada el cual debe ejecutarse el programa, y el número de tarea—en la implementación, es el programa el que devuelve el número de tarea (ver más abajo).

Las designaciones para los requerimientos son las siguientes:

Un usuario solicita ejecutar el programa p cada t unidades de tiempo, desde el momento en que se ejecuta esta acción, con el número de tarea $i \approx AddJob(t, p, i)$

Un usuario solicita eliminar la tarea programada $i \approx RemoveJob(i)$

Se solicita al sistema operativo que ejecute el programa asociado con la tarea programada número $i \approx Exec(i)$

Sin embargo, como especificamos los requisitos temporales a través de temporizadores, incluimos las siguientes designaciones:

Se inicia el temporizador i (asociado a la tarea i) $\approx Start(i)$

El temporizador i (asociado a la tarea i) comunica que ha transcurrido el tiempo establecido $\approx Sched(i)$

3. La especificación

Para especificar el sistema `cron` usamos la variable $crontab$ que mantiene el estado de cada una de las tareas que deben ser ejecutadas. Este estado va cambiando según transcurra el tiempo, según los pedidos efectuados por los usuarios y según la disponibilidad del sistema operativo para ejecutar un proceso. Como se muestra en la Figura 4 a cada tarea programada se le asocia un temporizador cuyo estado se registra en la varible $timers$. La variable $aprocs$ representa a los procesos que el sistema operativo está ejecutando en cada instante; se asume que esta variable puede ser modificada por otros módulos no especificados. Dado que el sistema operativo puede ejecutar simultaneamente una cantidad acotada de procesos, $aprocs$ es un conjunto de tamaño $MAXAPROCS$.

$crontab$ guarda, para cada tarea programada, el tiempo que debe transcurrir desde que un usuario suscribió la tarea hasta que se la ejecuta ($time$), el programa que se debe ejecutar llegado ese tiempo ($prog$) y el estado de la tarea ($status$). Los dos primeros valores son indicados por el usuario que da de alta la tarea al ejecutar la operación $AddJob$; el restante lo va modificando el sistema como se explica más abajo. Dado que el modelo nos permite abstraernos de los detalles de implementación, $crontab$ es una función con dominio en los naturales lo que permite que los usuarios registren una cantidad infinita de tareas programadas. El sistema parte de un estado en el que no hay ninguna tarea programada. Convenimos en que la tarea i no está programada si $aprocs[i].prog$ es $nullp$ y $aprocs[i].status$ es "none" (ver Figura 5).

Notar que $nullp$ fue elegido de manera tal que no pertenezca al conjunto de los programas del sistema, $PROGS$. Esto implica que el campo $prog$ puede almacenar tanto un elemento de $PROGS$ como algo que no pertenece a tal conjunto. En TLA esto se puede expresar de manera sencilla en virtud de que las variables no son tipadas. Para poder expresar lo mismo, digamos, en Z deberíamos haber recurrido a un tipo libre (podríamos llamarlo $APROGS$) para el campo $prog$ con dos constructores: uno constante, $nullp$; y otro, $rprog$, con dominio en $PROGS$:

$$APROGS ::= nullp \mid rprog \ll PROGS \gg$$

Por lo tanto, cada vez que tuviéramos algo de tipo $APROGS$ deberíamos preguntarnos si es $nullp$ o $rprog(p)$ para algún $p \in PROGS$, lo que torna la especificación más larga y compleja. Por

module *CronSys*

extends *Naturals, FiniteSets*
CONSTANTS *MAXAPROCS, PROGS*
ASSUME $MAXAPROCS \in Nat \land MAXAPROCS > 0$
VARIABLES *crontab, timers, now, aprocs*

$$
\begin{array}{lcl}
nullp & \triangleq & \text{CHOOSE } x : x \notin PROGS \\
TypeInv & \triangleq & \land\ timers \in [Nat \to [t, l : Nat, r : \{\text{“yes”}, \text{“no”}\}]] \\
 & & \land\ IsFiniteSet(aprocs) \\
 & & \land\ crontab \in [Nat \to [time : Nat, \\
 & & \qquad\qquad prog : PROGS \cup \{nullp\}, \\
 & & \qquad\qquad status : \{\text{“none”}, \text{“no”}, \text{“yes”}, \text{“run”}\}]] \\
 & & \land\ now \in Nat \\
Ts & \triangleq & \textbf{instance } Timers \\
av & \triangleq & \langle contrab, aprocs, timers, now \rangle \\
sv & \triangleq & \langle contrab, aprocs, timers \rangle
\end{array}
$$

Figura 4: Primera parte de la especificación.

ejemplo, el predicado $aprocs' = aprocs \cup \{crontab[i].prog\}$ que aparece en la definición de $Exec$ (ver Figura 5), en la especificación Z tendría que haber sido:

$$aprocs' = aprocs \cup \{p : PROGS \mid rprog\ p = (crontab\ i).prog \bullet p\}$$

En tanto que no podemos escribir simplemente $aprocs' = aprocs \cup \{(crontab\ i).prog\}$ puesto que $(crontab\ i).prog$ no es de tipo $PROGS$ aunque en ese momento $prog$ sea tal que existe un p en $PROGS$ tal que $rprog\ p = (crontab\ i).prog$.

En la Figura 5 podemos ver las operaciones del sistema. $AddJob$ recibe tres parámetros que debe suministrar el usuario. En la implementación el tercer parámentro podría ser generado por el sistema; queda como ejercicio para el alumno especificar la operación de tal forma. Notar que en la misma operación se establece el límite para el temporizador asociado con la tarea i. Esto puede ser confuso puesto parece que el entorno configura el temporizador cuando esto es una tarea del sistema; queda como ejercicio para el alumno modificar la especificación en este sentido; pueden consultar [2, páginas 12 a 18]. En este paso se pone la tarea a “no” lo que indica que la tarea está programada pero su temporizador no ha sido iniciado aun. La operación $Start$ es una de las operaciones internas del sistema. Se utiliza para iniciar el temporizador asociado con la tarea i que recibe como parámetro; notar que el sistema cambia el $status$ para esa tarea a “yes” lo que significa que el temporizador está corriendo.

Una vez que el temporizador asociado a una tarea ha sido iniciado el sistema deberá esperar a que se produzca el $Timeout$ correspondiente. Cuando esto ocurra el sistema cambiará el estado de la tarea a “run” indicando así que la tarea está lista para ser ejecutada. Todo esto está expresado en la operación $Sched$. Esta operación está controlada por el entorno dado que la acción $Timeout$ lo está. Aquí también cabe preguntarse si no sería conveniente incluir en $Sched$ la ejecución de la tarea i. Sea como fuere, llega el momento en que una tarea está lista para ser

$$
\begin{array}{l}
\mathit{Init} \triangleq \land\ Ts!TInit \\
\qquad\qquad \land\ crontab = [x \in Nat \mapsto [time \mapsto 0, prog \mapsto nullp, status \mapsto \text{“none”}]] \\
\mathit{AddJob(t,p,i)} \triangleq \land\ crontab[i].status = \text{“none”} \\
\qquad\qquad\qquad\qquad \land\ p \in PROGS \\
\qquad\qquad\qquad\qquad \land\ t > 0 \\
\qquad\qquad\qquad\qquad \land\ Ts!Set(i,t) \\
\qquad\qquad\qquad\qquad \land\ crontab' = [crontab \text{ EXCEPT } ![i] = [time \mapsto t, prog \mapsto p, status \mapsto \text{“no”}]] \\
\qquad\qquad\qquad\qquad \land\ \text{UNCHANGED } \langle aprocs, now \rangle \\
\mathit{Start(i)} \triangleq \land\ crontab[i].status = \text{“no”} \\
\qquad\qquad\qquad \land\ Ts!Start(i) \\
\qquad\qquad\qquad \land\ crontab' = \\
\qquad\qquad\qquad\qquad [crontab \text{ EXCEPT } ![i] = [time \mapsto @.time, prog \mapsto @.prog, status \mapsto \text{“yes”}]] \\
\qquad\qquad\qquad \land\ \text{UNCHANGED } \langle aprocs, now \rangle \\
\mathit{Sched(i)} \triangleq \land\ crontab[i].status = \text{“yes”} \\
\qquad\qquad\qquad\ \land\ Ts!Timeout(i) \\
\qquad\qquad\qquad\ \land\ crontab' = \\
\qquad\qquad\qquad\qquad [crontab \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad\qquad\qquad [time \mapsto @.time, prog \mapsto @.prog, status \mapsto \text{“run”}]] \\
\qquad\qquad\qquad\ \land\ \text{UNCHANGED } \langle aprocs, now \rangle \\
\mathit{Exec(i)} \triangleq \land\ crontab[i].status = \text{“run”} \\
\qquad\qquad\qquad \land\ Cardinality(aprocs) < MAXAPROCS \\
\qquad\qquad\qquad \land\ crontab' = \\
\qquad\qquad\qquad\qquad [crontab \text{ EXCEPT } ![i] = [time \mapsto @.time, prog \mapsto @.prog, status \mapsto \text{“no”}]] \\
\qquad\qquad\qquad \land\ aprocs' = aprocs \cup \{crontab[i].prog\} \\
\qquad\qquad\qquad \land\ \text{UNCHANGED } \langle timers, now \rangle \\
\mathit{RemoveJob(i)} \triangleq \land \lor \land\ crontab[i].status = \text{“yes”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ Ts!Stop(i) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ \text{UNCHANGED } \langle aprocs, now \rangle \\
\qquad\qquad\qquad\qquad\qquad\quad \lor \land\ crontab[i].status = \text{“no”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ \text{UNCHANGED } \langle aprocs, now, timers \rangle \\
\qquad\qquad\qquad\qquad\ \land\ crontab' = \\
\qquad\qquad\qquad\qquad\qquad [crontab \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad\qquad\qquad\qquad [time \mapsto 0, prog \mapsto nullp, status \mapsto \text{“none”}]] \\
\hline
\mathit{Next} \triangleq \exists\, i \in Nat : \lor\ Start(i) \lor Sched(i) \lor Exec(i) \lor RemoveJobe(i) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor (\exists\, t \in Nat, p \in PROGS : AddJob(t,p,i)) \\
\mathit{Spec} \triangleq Init \land \Box[Next]_{sv} \land (\forall\, i \in Nat : WF_{av}(Start(i)) \land SF_{av}(Exec(i))) \\
\hline
\text{THEOREM } Spec \Rightarrow \Box TypeInv
\end{array}
$$

Figura 5: Última parte de la especificación.

ejecutada. En este caso la operación interna $Exec$ verifica que el sistema operativo tenga recursos como para ejecutar un proceso más en cuyo caso esto se lleva a cabo; el caso contrario no está especificado.

La última operación es la que permite a un usuario remover una tarea programada. Aquí hemos decidido que se puede remover una tarea si está en estado "yes" o "no" pues nos parece incorrecto permitir la eliminación una vez que ya está lista para ser ejecutada (estado "run").

Yendo a la parte final de la especificación, vemos que lo más interesante son las fórmulas de equidad para cada operación. En primer término sólo hay que especificar requisitos de equidad para las operaciones internas que en este caso son $Start$ y $Exec$. Luego debemos tener en cuenta que siendo acciones parametrizadas $Start(i)$ y $Exec(i)$ deben considerarse como operaciones independientes para cada i; esto significa que tendremos una fórmula de equidad para cada una de ellas.

¿Deben ser fórmulas de equidad débil o fuerte? Como explica Lamport, SF y WF son equivalentes para una acción dada si siempre que esta está habilitada solo puede deshabilitarse al ser ejecutada. Además, Lamport indica que siempre debe preferirse equidad débil por sobre equidad fuerte si no hay un motivo importante por el cual la segunda opción sea la mejor. En términos más generales, se debe elegir entre equidad débil o fuerte en función del requerimiento que haya en ese sentido. Por ejemplo, si una acción controlada por el entorno puede deshabilitar a una acción interna, dependerá del requisito si se solicita equidad fuerte o débil para la acción interna (aunque en este caso débil y fuerte no son equivalentes). En efecto, si la acción externa provee datos que la acción interna debe procesar y el requisito es que si el entorno provee infinitos datos, el sistema debe procesar infinitos datos, entonces a la acción interna se le debe pedir equidad fuerte. Pero si la acción externa es un comando del usuario que intenta detener el procesamiento de ciertos datos, entonces equidad fuerte no tiene sentido y puede ser contraproducente.

Entonces según todas las consideraciones anteriores, vemos que $Start$ solo puede deshabilitarse a sí misma o por la ejecución de una operación externa ($RemoveJob$) por lo que $WF(Start(i))$ y $SF(Start(i))$ no son equivalentes en este caso. Pero la acción externa es un comando de usuario que se aplica a una tarea particular por lo que un requerimiento de SF para $Start(i)$ hubiera dado mayor equidad pero para otra tarea (el parámetro i puede ser el mismo pero si la primera tarea fue removida, la segunda vez que se use i la tarea será otra). En consecuencia optamos por equidad débil para $Start(i)$ para cada $i \in Nat$.

La situación es diferente para $Exec(i)$ pues $Exec(j)$ para $i \neq j$ puede deshabilitar a $Exec(i)$ dado que al ejecutar el programa correspondiente a la tarea j puede ocurrir que el sistema operativo se quede sin recursos para ejecutar el programa de la tarea i—es decir $Cardinality(aprocs) < MAXAPROCS$ podría ser falsa en $Exec(i)$. Es decir, $Exec(i)$ no solo se deshabilita a sí misma sino que también puede ser deshabilitada por otra operación interna—$Exec(j)$; observar que aquí queda bien claro que $Exec(i)$ y $Exec(j)$ son acciones distintas si $i \neq j$. Por lo tanto, requerimos SF para cada $Exec(i)$. De esta forma, si por alguna razón la tarea programada i (lista para ser ejecutada) es relegada reiteradas veces por la ejecución de otras tareas, llegará un momento en el cual el sistema ejecutará i sí o sí. Esto puede lograrse mediante un *scheduler* de prioridad el cual suba la prioridad de cada tarea i lista para ser ejecutada cada vez que no es ejecutada.

Referencias

[1] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] M. Abadi and S. Merz, "On TLA as a logic," in *Proceedings of the NATO Advanced Study Institute on Deductive program design*. Secaucus, NJ, USA: Springer-Verlag New York Inc., 1996, pp. 235–271.