Universidad Nacional de Rosario
Facultad de Ciencias Exactas,
Ingeniería y Agrimensura
Departamento de Ciencias de la
Computación

Ingeniería de Software

Trabajo Práctico
Verificación de Software

Cerruti Lautaro

Julio de 2025

1. Problema: Readers-Writer Lock

Se desea especificar el funcionamiento de la primitiva de sincronización Readers-Writer Lock para procesos.

El lock tiene que permitir múltiples lectores en simultáneo pero acceso exclusivo para la escritura. Adicionalmente se quiere poder establecer la cantidad máxima de lectores que se desea permitir en simultáneo con un mínimo de un lector.

Se deben especificar las operaciones:

- Adquirir lectura: si no hay ningún proceso escribiendo y no se llegó a máximo de lectores, se adquiere el lock en modo lectura.
- Soltar lectura: se suelta el lock previamente adquirido en modo lectura.
- Adquirir escritura: si no hay ningún proceso leyendo ni escribiendo, se adquiere el lock en modo escritura.
- Soltar escritura: se suelta el lock previamente adquirido en modo escritura.
- Establecer máximo de lectores: se establece un nuevo valor máximo de cantidad de lectores en simultáneo.

2. Especificación

Primero veamos las designaciones:

p es un proceso $\approx p \in PROCESS$

creatorProcess es el proceso que crea el lock $\approx creatorProcess \in PROCESS$

state es el estado de escritura del lock $\approx state \in STATE$

res es un mensaje de respuesta $\approx res \in RESPONSE$

el estado del lock $\approx ReadersWriterLock$

el estado inicial del lock $\approx ReadersWriterLockInit$

se establece una cantidad maxima de lectores $\approx SetMaxReaders$

se obtiene el lock en modo lectura $\approx AcquireRead$

se suelta el lock en modo lectura $\approx ReleaseRead$

se obtiene el lock en modo escritura $\approx AcquireWrite$

se suelta el lock en modo escritura $\approx ReleaseWrite$

Ahora definamos los tipos básicos y enumerados.

$[PROCESS]$

$STATE ::= locked \mid unlocked$

$RESPONSE ::= ok \mid errorCantBeLessThanOne \mid errorCantAllowMoreReaders$
$\mid errorCantBeLessThanActualReaders \mid errorLockedByWriter \mid errorLockedByReader$
$\mid errorReadNotAcquired \mid errorWriteNotLocked \mid errorWriteLockedByOtherProcess$
$\mid errorAlreadyAcquired$

Definimos de forma axiomática un proceso llamado *creatorProcess* para establecer como valor inicial de la variable de estado *writer*.

$creatorProcess : PROCESS$

Luego definimos el espacio de estado del lock y el estado inicial:

ReadersWriterLock

$$readers : \mathbb{P}\, PROCESS$$
$$writerLockState : STATE$$
$$writer : PROCESS$$
$$maxReaders : \mathbb{Z}$$

ReadersWriterLockInit

ReadersWriterLock

$$readers = \emptyset$$
$$writerLockState = unlocked$$
$$writer = creatorProcess$$
$$maxReaders = 1$$

Cabe destacar que el valor de la variable de estado *writer* solo es relevante cuando el estado del lock *writerLockState* se encuentra en *locked*. Es necesario inicializarla debido a que la misma tiene que tener un valor para poder hacer las simulaciones en NEXT, la inicialización fue agregada a la especificación por cuestiones de consistencia entre esta y su traducción a $\{log\}$.

Vamos a definir tres invariantes.

La primera establece que la cantidad máxima de lectores siempre tiene que ser mayor a cero:

InvMaxReadersPositive

ReadersWriterLock

$$maxReaders > 0$$

La segunda representa que la cantidad de lectores nunca puede ser mayor al máximo establecido:

InvReadersLessThanMaxReaders

ReadersWriterLock

$$\#readers \leq maxReaders$$

Y la última establece que si hay algún proceso escribiendo no puede haber ningún proceso leyendo:

InvNoReadersWhileWriter

$$ReadersWriterLock$$

$$writerLockState = locked \Rightarrow readers = \emptyset$$

La primera operación a especificar es *SetMaxReaders*. En el caso exitoso, donde el valor que se intenta establecer es válido, se guarda el nuevo máximo y se comunica que la operación se pudo realizar correctamente.

SetMaxReadersOk

$$\Delta ReadersWriterLock$$
$$n? : \mathbb{Z}$$
$$res! : RESPONSE$$

$$n? > 0$$
$$\#readers \leq n?$$
$$maxReaders' = n?$$
$$readers' = readers$$
$$writerLockState' = writerLockState$$
$$writer' = writer$$
$$res! = ok$$

Para esta operación tenemos dos casos de error, el primero es si el valor que se está tratando de establecer como máximo es menor a uno y el segundo se da cuando se está tratando de establecer un máximo menor a la cantidad de lectores actual.

$$
\begin{array}{|l}
\hline
MaxReadersIncorrectValue \\
\hline
\Xi ReadersWriterLock \\
n? : \mathbb{Z} \\
res! : RESPONSE \\
\hline
n? \leq 0 \\
res! = errorCantBeLessThanOne \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
LessThanActualReaders \\
\hline
\Xi ReadersWriterLock \\
n? : \mathbb{Z} \\
res! : RESPONSE \\
\hline
n? < \#readers \\
res! = errorCantBeLessThanActualReaders \\
\hline
\end{array}
$$

$$SetMaxReadersErrors \mathrel{\widehat=} MaxReadersIncorrectValue \lor LessThanActualReaders$$

$$SetMaxReaders \mathrel{\widehat=} SetMaxReadersOk \lor SetMaxReadersErrors$$

Ahora veamos la operación para adquirir el lock en modo lectura. En el caso exitoso se agrega al proceso como lector y se responde con un mensaje indicando que se pudo obtener el lock.

AcquireReadOk

$$\Delta ReadersWriterLock$$
$$p? : PROCESS$$
$$res! : RESPONSE$$

$$p? \notin readers$$
$$writerLockState = unlocked$$
$$\#readers < maxReaders$$
$$readers' = readers \cup \{p?\}$$
$$writerLockState' = writerLockState$$
$$writer' = writer$$
$$maxReaders' = maxReaders$$
$$res! = ok$$

Para esta operación tenemos tres casos de error: ya se había adquirido previamente, ya se llegó al máximo de lectores permitidos o hay algún proceso escribiendo.

AlreadyLockedRead

$$\Xi ReadersWriterLock$$
$$p? : PROCESS$$
$$res! : RESPONSE$$

$$p? \in readers$$
$$res! = errorAlreadyAcquired$$

MaxReadersReached

$$\Xi ReadersWriterLock$$
$$res! : RESPONSE$$

$$\#readers = maxReaders$$
$$res! = errorCantAllowMoreReaders$$

ProcessIsWriting

$\Xi ReadersWriterLock$

$res! : RESPONSE$

$writerLockState = locked$

$res! = errorLockedByWriter$

$AcquireReadError \mathrel{\widehat=} AlreadyLockedRead \lor ProcessIsWriting \lor MaxReadersReached$

$AcquireRead \mathrel{\widehat=} AcquireReadOk \lor AcquireReadError$

Continuemos con la operación para adquirir el lock en modo escritura. En el caso exitoso se adquiere el lock correctamente, se establece el proceso que lo adquirió como el escritor y se retorna una respuesta informando la correcta adquisición.

AcquireWriteOk

$\Delta ReadersWriterLock$

$p? : PROCESS$

$res! : RESPONSE$

$writerLockState = unlocked$

$readers = \emptyset$

$writer' = p?$

$writerLockState' = locked$

$readers' = readers$

$maxReaders' = maxReaders$

$res! = ok$

Nuevamente tenemos tres posibles casos de error: ya se había adquirido el lock previamente, ya hay algún otro proceso escribiendo o hay algún proceso leyendo.

AlreadyLockedWrite

$\Xi ReadersWriterLock$

$p? : PROCESS$

$res! : RESPONSE$

$writerLockState = locked$

$writer = p?$

$res! = errorAlreadyAcquired$

WriteLockedByOtherProcess

$\Xi ReadersWriterLock$

$p? : PROCESS$

$res! : RESPONSE$

$writerLockState = locked$

$writer \neq p?$

$res! = errorWriteLockedByOtherProcess$

ProcessIsReading

$\Xi ReadersWriterLock$

$res! : RESPONSE$

$readers \neq \emptyset$

$res! = errorLockedByReader$

$$AcquireWriteError \mathrel{\widehat{=}} AlreadyLockedWrite \lor WriteLockedByOtherProcess \lor ProcessIsReading$$

$$AcquireWrite \mathrel{\widehat{=}} AcquireWriteOk \lor AcquireWriteError$$

Ahora veamos la operación para soltar el lock previamente adquirido en modo lectura. En el caso exitoso se saca al proceso que soltó el lock de los lectores actuales.

ReleaseReadOk

$\Delta ReadersWriterLock$

$p? : PROCESS$

$res! : RESPONSE$

$p? \in readers$

$readers' = readers \setminus \{p?\}$

$writerLockState' = writerLockState$

$writer' = writer$

$maxReaders' = maxReaders$

$res! = ok$

Para esta operación solo consideramos un caso de error el cual sucede cuando el proceso que está intentando soltar el lock nunca lo adquirió.

ReadNotAcquired

$\Xi ReadersWriterLock$

$p? : PROCESS$

$res! : RESPONSE$

$p? \notin readers$

$res! = errorReadNotAcquired$

$ReleaseRead \mathrel{\widehat=} ReleaseReadOk \lor ReadNotAcquired$

Por último tenemos la operación para soltar el lock adquirido en modo escritura. En su caso exitoso se suelta el lock y se establece el estado en consecuencia.

ReleaseWriteOk

$$\Delta ReadersWriterLock$$
$$p? : PROCESS$$
$$res! : RESPONSE$$

$$writerLockState = locked$$
$$writer = p?$$
$$writerLockState' = unlocked$$
$$readers' = readers$$
$$writer' = writer$$
$$maxReaders' = maxReaders$$
$$res! = ok$$

Y tenemos dos posibles casos de error: uno donde el lock esté con estado *unlocked*, es decir que ningún proceso lo haya tomado en modo escritura y otro donde el lock fue adquirido en modo escritura por un proceso distinto al que lo está tratando de soltar.

WriteNotLocked

$$\Xi ReadersWriterLock$$
$$res! : RESPONSE$$

$$writerLockState = unlocked$$
$$res! = errorWriteNotLocked$$

$$ReleaseWriteError \mathrel{\widehat{=}} WriteNotLocked \lor WriteLockedByOtherProcess$$
$$ReleaseWrite \mathrel{\widehat{=}} ReleaseWriteOk \lor ReleaseWriteError$$

La especificación pasa el control de tipos de Fuzz, se adjunta la misma en un archivo *.tex* de forma individual para que esto pueda ser verificado, adicionalmente podemos encontrar en el archivo **readers-writer-lock-fuzz-output** la salida de fuzz al ejecutarlo con la opción $-t$ con la que podemos verificar los tipos de toda la especificación.

3. Simulaciones

Las simulaciones fueron divididas en dos situaciones, la primera que representa un uso más real del sistema y una segunda para probar todos los casos de errores de las distintas operaciones.

Para ejecutar estas simulaciones se cargó el archivo de la traducción a un programa $\{log\}$, se corrió el VCG, se cargó el archivo que generó el VCG y se ejecutaron las distintas simulaciones.

3.1. Primera simulación

Esta simulación es la que busca evaluar un escenario funcional realista donde si bien no se prueban todos los casos de error, se utilizan todas las operaciones.

En la misma probamos establecer un máximo de lectores, que varios procesos adquieran el lock en modo lectura, que un proceso intente adquirirlo en modo lectura cuando ya se llegó al máximo de lectores, que un proceso intente adquirir en modo escritura cuando hay procesos como lectores, adquisiciones exitosas en modo escritura y que el lock sea soltado en ambos modos.

```
[initial]:[invMaxReadersPositive, invReadersLessThanMaxReaders,
    invNoReadersWhileWriter]
    >> setMaxReaders(N:2)
    >> acquireRead(P:p1)
    >> acquireRead(P:p2)
    >> acquireRead(P:p3)
    >> acquireWrite(P:p3)
    >> releaseRead(P:p1)
    >> acquireRead(P:p1)
    >> releaseRead(P:p1)
    >> releaseRead(P:p2)
    >> acquireWrite(P:p3)
    >> acquireWrite(P:p2)
    >> acquireRead(P:p1)
    >> releaseWrite(P:p3).
```

Esta simulación generó la siguiente traza:

```
Execution trace is:
Readers = {},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 1
  ----> setMaxReaders(N:2)
Res = ok,
Readers = {},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireRead(P:p1)
Res = ok,
Readers = {p1},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireRead(P:p2)
Res = ok,
Readers = {p1,p2},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireRead(P:p3)
Res = errorCantAllowMoreReaders,
Readers = {p1,p2},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireWrite(P:p3)
Res = errorLockedByReader,
```

```
Readers = {p1,p2},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> releaseRead(P:p1)
Res = ok,
Readers = {p2},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireRead(P:p1)
Res = ok,
Readers = {p2,p1},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> releaseRead(P:p1)
Res = ok,
Readers = {p2},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> releaseRead(P:p2)
Res = ok,
Readers = {},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireWrite(P:p3)
Res = ok,
Readers = {},
```

```
WriterLockState = locked,
Writer = p3,
MaxReaders = 2
  ----> acquireWrite(P:p2)
Res = errorWriteLockedByOtherProcess,
Readers = {},
WriterLockState = locked,
Writer = p3,
MaxReaders = 2
  ----> acquireRead(P:p1)
Res = errorLockedByWriter,
Readers = {},
WriterLockState = locked,
Writer = p3,
MaxReaders = 2
  ----> releaseWrite(P:p3)
Res = ok,
Readers = {},
WriterLockState = unlocked,
Writer = p3,
MaxReaders = 2

true
```

Se obtuvo el resultado esperado y se verificaron las invariantes en toda la simulación.

3.2. Segunda simulación

En esta segunda simulación buscamos probar que todos los casos de error que no probamos con la simulación anterior verifiquen los invariantes. Esto implica probar cosas como establecer un máximo de lectores menor a los lectores actuales, establecer un valor máximo inválido, soltar el lock en ambos modos sin haberlo adquirido previamente, tratar de adquirir un lock cuando ya había sido adquirido, tratar de soltar un lock adquirido por otro proceso y tratar de soltar un

lock que no fue adquirido por ningún proceso.

La simulación en la siguiente:

```
[initial]:[invMaxReadersPositive, invReadersLessThanMaxReaders,
    invNoReadersWhileWriter]
    >> setMaxReaders(N:0)
    >> setMaxReaders(N:2)
    >> acquireRead(P:p1)
    >> acquireRead(P:p2)
    >> setMaxReaders(N:1)
    >> releaseRead(P:p2)
    >> releaseRead(P:p2)
    >> acquireRead(P:p1)
    >> releaseRead(P:p1)
    >> acquireWrite(P:p1)
    >> acquireWrite(P:p1)
    >> releaseWrite(P:p2)
    >> releaseWrite(P:p1)
    >> releaseWrite(P:p1).
```

La cual genera la siguiente traza:

```
Execution trace is:
Readers = {},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 1
  ----> setMaxReaders(N:0)
Res = errorCantBeLessThanOne,
Readers = {},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 1
```

```
  ----> setMaxReaders(N:2)
Res = ok,
Readers = {},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireRead(P:p1)
Res = ok,
Readers = {p1},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireRead(P:p2)
Res = ok,
Readers = {p1,p2},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> setMaxReaders(N:1)
Res = errorCantBeLessThanActualReaders,
Readers = {p1,p2},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> releaseRead(P:p2)
Res = ok,
Readers = {p1},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> releaseRead(P:p2)
```

```
Res = errorReadNotAcquired,
Readers = {p1},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireRead(P:p1)
Res = errorAlreadyAcquired,
Readers = {p1},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> releaseRead(P:p1)
Res = ok,
Readers = {},
WriterLockState = unlocked,
Writer = creatorProcess,
MaxReaders = 2
  ----> acquireWrite(P:p1)
Res = ok,
Readers = {},
WriterLockState = locked,
Writer = p1,
MaxReaders = 2
  ----> acquireWrite(P:p1)
Res = errorAlreadyAcquired,
Readers = {},
WriterLockState = locked,
Writer = p1,
MaxReaders = 2
  ----> releaseWrite(P:p2)
Res = errorWriteLockedByOtherProcess,
```

```
Readers = {},
WriterLockState = locked,
Writer = p1,
MaxReaders = 2
  ----> releaseWrite(P:p1)
Res = ok,
Readers = {},
WriterLockState = unlocked,
Writer = p1,
MaxReaders = 2
  ----> releaseWrite(P:p1)
Res = errorWriteNotLocked,
Readers = {},
WriterLockState = unlocked,
Writer = p1,
MaxReaders = 2

true
```

Al igual que con la simulación anterior, se puede observar que la traza es correcta y que se verifican las invariantes de estado.

4. VCG

La ejecución del generador de condiciones de verificación (VCG) se realizó con el siguiente comando:

```
vcg('reader-writer-lock.slog').
```

y este nos generó el archivo **reader-writer-lock-vc.slog**. Consultando el mismo y ejecutando la verificación con los siguientes comandos:

```
consult('reader-writer-lock-vc.slog').
check_vcs_reader-writer-lock.
```

Tenemos el siguiente resultado:

```
Checking readersWriterLockInit_sat_invMaxReadersPositive ... OK
Checking readersWriterLockInit_sat_invReadersLessThanMaxReaders ... OK
Checking readersWriterLockInit_sat_invNoReadersWhileWriter ... OK
Checking setMaxReaders_is_sat ... OK
Checking acquireRead_is_sat ... OK
Checking acquireWrite_is_sat ... OK
Checking releaseRead_is_sat ... OK
Checking releaseWrite_is_sat ... OK
Checking setMaxReaders_pi_invMaxReadersPositive ... OK
Checking setMaxReaders_pi_invReadersLessThanMaxReaders ... OK
Checking setMaxReaders_pi_invNoReadersWhileWriter ... OK
Checking acquireRead_pi_invMaxReadersPositive ... OK
Checking acquireRead_pi_invReadersLessThanMaxReaders ... OK
Checking acquireRead_pi_invNoReadersWhileWriter ... OK
Checking acquireWrite_pi_invMaxReadersPositive ... OK
Checking acquireWrite_pi_invReadersLessThanMaxReaders ... OK
Checking acquireWrite_pi_invNoReadersWhileWriter ... OK
Checking releaseRead_pi_invMaxReadersPositive ... OK
Checking releaseRead_pi_invReadersLessThanMaxReaders ... OK
Checking releaseRead_pi_invNoReadersWhileWriter ... OK
Checking releaseWrite_pi_invMaxReadersPositive ... OK
Checking releaseWrite_pi_invReadersLessThanMaxReaders ... OK
Checking releaseWrite_pi_invNoReadersWhileWriter ... OK

Total VCs: 23 (discharged: 23, failed: 0, timeout: 0)
Execution time (discharged): 0.09174370765686035 s
yes
```

Podemos ver que todas las condiciones son verificadas correctamente por lo que no fue necesario agregar ninguna hipótesis ni utilizar los comandos *vcgce*, *vcacg* o *findh*.

5. TTF

La generación de los casos de prueba se realizó para cada operación en particular y corresponde analizar cada una por separado, ya que son distintas las tácticas aplicadas. En todas se comenzó aplicando DNF para luego aplicar otras tácticas sobre algunas de las clases de prueba generadas.

5.1. SetMaxReaders

Luego de aplicar DNF notamos que la primera clase de la disyunción corresponde al caso de *SetMaxReadersOk*, la segunda corresponde a *maxReadersIncorrectValue* donde el valor que se está tratando de establecer como máximo es menor a 1 y el tercer caso de la disyunción corresponde a *lessThanActualReaders* donde se esta tratando de establecer un máximo menor a la cantidad de lectores actuales.

Por esto se decidió primero aplicarle cardinalidad de conjuntos a la clase **setMaxReaders_dnf_1** para generar casos donde se establezcan distintos valores de máximo cuando hay lectores y cuando no. Luego aplicamos intervalos de enteros a **setMaxReaders_dnf_2** para que se generen casos donde se prueban más valores que simplemente 0 como valores de máximo inválidos. Por último aplicamos intervalo de enteros a **setMaxReaders_dnf_3** para que se generen casos donde hay más lectores.

Los comandos utilizados para generar los casos de prueba para esta operación fueron:

```
ttf(rwlock).
applydnf(setMaxReaders(N)).
applysc(setMaxReaders_dnf_1,Readers).
applyii(setMaxReaders_dnf_2,N,[-1,0]).
applyii(setMaxReaders_dnf_3,N,[2,3]).
prunett.
gentc.
exporttt.
```

El ejecutar *writett.* para ver el árbol de pruebas generado tenemos:

```
setMaxReaders_vis
  setMaxReaders_dnf_1
```

```
    setMaxReaders_sc_11 -> setMaxReaders_tc_11
    setMaxReaders_sc_12 -> setMaxReaders_tc_12
    setMaxReaders_sc_13 -> setMaxReaders_tc_13
  setMaxReaders_dnf_2
    setMaxReaders_ii_21 -> setMaxReaders_tc_21
    setMaxReaders_ii_22 -> setMaxReaders_tc_22
    setMaxReaders_ii_24 -> setMaxReaders_tc_24
  setMaxReaders_dnf_3
    setMaxReaders_ii_31 -> setMaxReaders_tc_31
    setMaxReaders_ii_32 -> setMaxReaders_tc_32
    setMaxReaders_ii_34 -> setMaxReaders_tc_34
    setMaxReaders_ii_35 -> setMaxReaders_tc_35
```

En el archivo **rwlock_acquireWrite-tt.slog** podemos encontrar los casos de prueba generados, el contenido del mismo es el siguiente:

```
setMaxReaders_ts_11(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N>=_107000 &
  dec(_107000,int) &
  size(Readers,_107000) &
  N>0 &
  Readers={}.

setMaxReaders_tc_11(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=1 &
  Readers={}.

setMaxReaders_ts_12(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N>=_107000 &
  dec(_107000,int) &
  size(Readers,_107000) &
  N>0 &
  Readers={_107496} &
```

```
  dec(_107496,sum([creatorProcess,u(processPrevAxiom)])).

setMaxReaders_tc_12(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=1 &
  Readers={u(processPrevAxiom:n0)}.

setMaxReaders_ts_13(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N>=_107000 &
  dec(_107000,int) &
  size(Readers,_107000) &
  N>0 &
  Readers={_107496,_107566/_107568} &
  _107496 neq _107566 &
  dec([_107496,_107566],sum([creatorProcess,u(processPrevAxiom)])) &
  dec(_107568,set(sum([creatorProcess,u(processPrevAxiom)]))).

setMaxReaders_tc_13(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=2 &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1)}.

setMaxReaders_ts_21(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=<0 &
  N< -1.

setMaxReaders_tc_21(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N= -2.

setMaxReaders_ts_22(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=<0 &
  N= -1.
```

```
setMaxReaders_tc_22(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N= -1.

setMaxReaders_ts_24(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=<0 &
  N=0.

setMaxReaders_tc_24(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=0.

setMaxReaders_ts_31(N,Readers,WriterLockState,Writer,MaxReaders) :-
  _107254>N &
  dec(_107254,int) &
  size(Readers,_107254) &
  N<2.

setMaxReaders_tc_31(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N= -1 &
  Readers={}.

setMaxReaders_ts_32(N,Readers,WriterLockState,Writer,MaxReaders) :-
  _107254>N &
  dec(_107254,int) &
  size(Readers,_107254) &
  N=2.

setMaxReaders_tc_32(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=2 &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1),u(processPrevAxiom:n2)}.

setMaxReaders_ts_34(N,Readers,WriterLockState,Writer,MaxReaders) :-
```

```
  _107254>N &
  dec(_107254,int) &
  size(Readers,_107254) &
  N=3.

setMaxReaders_tc_34(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=3 &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1),u(processPrevAxiom:n2),
    u(processPrevAxiom:n3)}.

setMaxReaders_ts_35(N,Readers,WriterLockState,Writer,MaxReaders) :-
  _107254>N &
  dec(_107254,int) &
  size(Readers,_107254) &
  3<N.

setMaxReaders_tc_35(N,Readers,WriterLockState,Writer,MaxReaders) :-
  N=4 &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1),u(processPrevAxiom:n2),
    u(processPrevAxiom:n3),u(processPrevAxiom:n4)}.
```

5.2. AcquireRead

Para esta operación aplicamos primero DNF, esto resulta en cuatro disyunciones, analicemos cada una de forma particular:

1. La primera disyunción corresponde al caso de *AcquireReadOk*, aquí nos pareció sensato aplicar como segunda táctica Partición Estándar sobre la unión (operador *un*), ya que la misma contiene la implementación más compleja de la operación.
2. La segunda corresponde a cuando el lock está adquirido en modo lectura.
3. La tercera evalúa cuando el proceso ya adquirió el lock en modo lectura previamente y está tratando de volverlo a adquirir, acá nos pareció interesante aplicar Cardinalidad de

Conjuntos para generar casos donde el proceso este como lector solo o con otro proceso.

4. La cuarta disyunción corresponde a cuando se alcanzó el máximo de lectores, para este caso aplicamos Intervalo de Enteros para generar casos donde la cantidad máxima toma valores distintos.

Toda la generación de casos de pruebas se realizó con la siguiente serie de comandos:

```
ttf(rwlock).
applydnf(acquireRead(P)).
applysp(acquireRead_dnf_1,un(Readers, {P}, Readers_)).
applysc(acquireRead_dnf_3,Readers).
applyii(acquireRead_dnf_4,MaxReaders,[1,2,3]).
prunett.
gentc.
exporttt.
```

El árbol de pruebas generados es el siguiente:

```
acquireRead_vis
  acquireRead_dnf_1
    acquireRead_sp_12 -> acquireRead_tc_12
    acquireRead_sp_14 -> acquireRead_tc_14
  acquireRead_dnf_2 -> acquireRead_tc_2
  acquireRead_dnf_3
    acquireRead_sc_32 -> acquireRead_tc_32
    acquireRead_sc_33 -> acquireRead_tc_33
  acquireRead_dnf_4
    acquireRead_ii_41 -> acquireRead_tc_41
    acquireRead_ii_42 -> acquireRead_tc_42
    acquireRead_ii_44 -> acquireRead_tc_44
    acquireRead_ii_46 -> acquireRead_tc_46
    acquireRead_ii_47 -> acquireRead_tc_47
```

Esto nos generó los casos de pruebas en el archivo **rwlock_acquireRead-tt.slog** donde

podemos encontrar lo siguiente:

```
acquireRead_ts_12(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders>_78706 &
  dec(_78706,int) &
  size(Readers,_78706) &
  WriterLockState=unlocked &
  P nin Readers &
  Readers={} &
  {P}neq{}.

acquireRead_tc_12(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=1 &
  Readers={} &
  WriterLockState=unlocked &
  P=u(processPrevAxiom:n0).

acquireRead_ts_14(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders>_78706 &
  dec(_78706,int) &
  size(Readers,_78706) &
  WriterLockState=unlocked &
  P nin Readers &
  Readers neq {} &
  {P}neq{} &
  disj(Readers,{P}).

acquireRead_tc_14(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=2 &
  Readers={u(processPrevAxiom:n1)} &
  WriterLockState=unlocked &
  P=u(processPrevAxiom:n0).
```

```
acquireRead_ts_2(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=locked.

acquireRead_tc_2(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=locked.

acquireRead_ts_32(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P in Readers &
  Readers={_79492} &
  dec(_79492,sum([creatorProcess,u(processPrevAxiom)])).

acquireRead_tc_32(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P=u(processPrevAxiom:n0) &
  Readers={u(processPrevAxiom:n0)}.

acquireRead_ts_33(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P in Readers &
  Readers={_79492,_79562/_79564} &
  _79492 neq _79562 &
  dec([_79492,_79562],sum([creatorProcess,u(processPrevAxiom)])) &
  dec(_79564,set(sum([creatorProcess,u(processPrevAxiom)]))).

acquireRead_tc_33(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P=u(processPrevAxiom:n0) &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1)}.

acquireRead_ts_41(P,Readers,WriterLockState,Writer,MaxReaders) :-
  _79082=MaxReaders &
  dec(_79082,int) &
  size(Readers,_79082) &
```

```
  MaxReaders<1.

acquireRead_tc_41(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=0 &
  Readers={}.

acquireRead_ts_42(P,Readers,WriterLockState,Writer,MaxReaders) :-
  _79082=MaxReaders &
  dec(_79082,int) &
  size(Readers,_79082) &
  MaxReaders=1.

acquireRead_tc_42(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=1 &
  Readers={u(processPrevAxiom:n0)}.

acquireRead_ts_44(P,Readers,WriterLockState,Writer,MaxReaders) :-
  _79082=MaxReaders &
  dec(_79082,int) &
  size(Readers,_79082) &
  MaxReaders=2.

acquireRead_tc_44(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=2 &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1)}.

acquireRead_ts_46(P,Readers,WriterLockState,Writer,MaxReaders) :-
  _79082=MaxReaders &
  dec(_79082,int) &
  size(Readers,_79082) &
  MaxReaders=3.
```

```
acquireRead_tc_46(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=3 &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1),u(processPrevAxiom:n2)}.

acquireRead_ts_47(P,Readers,WriterLockState,Writer,MaxReaders) :-
  _79082=MaxReaders &
  dec(_79082,int) &
  size(Readers,_79082) &
  3<MaxReaders.

acquireRead_tc_47(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=4 &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1),u(processPrevAxiom:n2),
    u(processPrevAxiom:n3)}.
```

Resulta importante destacar el siguiente caso de prueba:

```
acquireRead_ts_41(P,Readers,WriterLockState,Writer,MaxReaders) :-
  _79082=MaxReaders &
  dec(_79082,int) &
  size(Readers,_79082) &
  MaxReaders<1.

acquireRead_tc_41(P,Readers,WriterLockState,Writer,MaxReaders) :-
  MaxReaders=0 &
  Readers={}.
```

En el mismo se evalúa adquirir el lock en modo lectura cuando la cantidad máxima de lectores es 0, este es un estado que viola el invariante *InvMaxReadersPositive*, probablemente este caso de prueba debería ser manualmente eliminado, ya que el programa no tendría ningún comportamiento definido.

5.3. AcquireWrite

Para esta operación nuevamente aplicamos DNF y analizamos las disyunciones generadas:

1. La primera disyunción corresponde al caso de *AcquireWriteOk*, aplicamos como segunda táctica Cardinalidad de Conjuntos sobre el conjunto de los lectores, esta clase de prueba *acquireWrite_dnf_1* tiene como condición que el conjunto de lectores esté vacío por lo que de los casos generados al aplicar CC solo debería ser satisfacible uno (donde $Readers = \{\}$). Al ejecutar *punett*. solo uno de los casos no es podado.
2. La segunda corresponde a cuando el proceso que está intentando obtener el lock ya lo obtuvo previamente en modo escritura.
3. La tercera se da cuando el lock fue obtenido por otro proceso en modo escritura.
4. La cuarta disyunción corresponde a cuando hay algún proceso leyendo. Aquí también aplicamos Cardinalidad de Conjuntos para generar casos de prueba donde haya más de un lector.

Toda la generación de casos de pruebas se realizó con la siguiente serie de comandos:

```
ttf(rwlock).
applydnf(acquireWrite(P)).
applysc(acquireWrite_dnf_1,Readers).
applysc(acquireWrite_dnf_4,Readers).
prunett.
gentc.
exporttt.
```

El árbol de pruebas generados es el siguiente:

```
acquireWrite_vis
  acquireWrite_dnf_1
    acquireWrite_sc_11 -> acquireWrite_tc_11
  acquireWrite_dnf_2 -> acquireWrite_tc_2
  acquireWrite_dnf_3 -> acquireWrite_tc_3
  acquireWrite_dnf_4
```

```
    acquireWrite_sc_42 -> acquireWrite_tc_42
    acquireWrite_sc_43 -> acquireWrite_tc_43
```

Esto nos generó los casos de pruebas en el archivo **rwlock_acquireWrite-tt.slog** donde podemos encontrar lo siguiente:

```
acquireWrite_ts_11(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Readers={} &
  WriterLockState=unlocked &
  Readers={}.

acquireWrite_tc_11(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Readers={} &
  WriterLockState=unlocked.

acquireWrite_ts_2(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer=P &
  WriterLockState=locked.

acquireWrite_tc_2(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer=u(processPrevAxiom:n0) &
  P=u(processPrevAxiom:n0) &
  WriterLockState=locked.

acquireWrite_ts_3(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer neq P &
  WriterLockState=locked.

acquireWrite_tc_3(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer=u(processPrevAxiom:n0) &
  P=u(processPrevAxiom:n1) &
  WriterLockState=locked.
```

```
acquireWrite_ts_42(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Readers neq {} &
  Readers={_68582} &
  dec(_68582,sum([creatorProcess,u(processPrevAxiom)])).

acquireWrite_tc_42(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Readers={u(processPrevAxiom:n0)}.

acquireWrite_ts_43(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Readers neq {} &
  Readers={_68582,_68652/_68654} &
  _68582 neq _68652 &
  dec([_68582,_68652],sum([creatorProcess,u(processPrevAxiom)])) &
  dec(_68654,set(sum([creatorProcess,u(processPrevAxiom)]))).

acquireWrite_tc_43(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1)}.
```

5.4. ReleaseRead

En esta operación al aplicar DNF solo tenemos dos disyunciones, una el caso de *ReleaseReadOk* y la otra es el caso de *ReadNotAcquired* donde se está tratando de soltar un lock que no fue adquirido previamente.
Para la primera disyunción aplicamos Partición Estándar sobre la diferencia de conjuntos $diff(Readers, P, Readers\_)$ que es la parte más importante de la operación.
Sobre el segundo caso generado por DNF aplicamos Cardinalidad de Conjuntos sobre los lectores para generar casos que evalúen que funciona correctamente independientemente de la cantidad de los mismos.

Los comandos utilizados para la generación de los casos de prueba fueron:

```
ttf(rwlock).
applydnf(releaseRead(P)).
```

```
applysp(releaseRead_dnf_1,diff(Readers, {P}, Readers_)).
applysc(releaseRead_dnf_2,Readers).
prunett.
gentc.
exporttt.
```

El árbol de pruebas es el siguiente:

```
releaseRead_vis
  releaseRead_dnf_1
    releaseRead_sp_16 -> releaseRead_tc_16
    releaseRead_sp_17 -> releaseRead_tc_17
  releaseRead_dnf_2
    releaseRead_sc_21 -> releaseRead_tc_21
    releaseRead_sc_22 -> releaseRead_tc_22
    releaseRead_sc_23 -> releaseRead_tc_23
```

Y los casos de prueba fueron generados en el archivo **rwlock_releaseRead-tt.slog**:

```
releaseRead_ts_16(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P in Readers &
  Readers neq {} &
  {P}neq{} &
  subset({P},Readers) &
  Readers neq {P}.

releaseRead_tc_16(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P=u(processPrevAxiom:n0) &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1)}.

releaseRead_ts_17(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P in Readers &
  Readers neq {} &
  {P}neq{} &
```

```
  Readers={P}.

releaseRead_tc_17(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P=u(processPrevAxiom:n0) &
  Readers={u(processPrevAxiom:n0)}.

releaseRead_ts_21(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P nin Readers &
  Readers={}.

releaseRead_tc_21(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P=u(processPrevAxiom:n0) &
  Readers={}.

releaseRead_ts_22(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P nin Readers &
  Readers={_64012} &
  dec(_64012,sum([creatorProcess,u(processPrevAxiom)])).

releaseRead_tc_22(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P=u(processPrevAxiom:n0) &
  Readers={u(processPrevAxiom:n1)}.

releaseRead_ts_23(P,Readers,WriterLockState,Writer,MaxReaders) :-
  P nin Readers &
  Readers={_64012,_64082/_64084} &
  _64012 neq _64082 &
  dec([_64012,_64082],sum([creatorProcess,u(processPrevAxiom)])) &
  dec(_64084,set(sum([creatorProcess,u(processPrevAxiom)]))).

releaseRead_tc_23(P,Readers,WriterLockState,Writer,MaxReaders) :-
```

```
  P=u(processPrevAxiom:n0) &
  Readers={u(processPrevAxiom:n1),u(processPrevAxiom:n2)}.
```

5.5. ReleaseWrite

Al aplicar DNF sobre esta operación obtenemos tres disyunciones. La primera corresponde al caso de *ReleaseWriteOk*, la segunda corresponde a cuando se está tratando de soltar un lock que no fue tomado por ningún proceso en modo lectura y la tercera a cuando el lock fue adquirido por un proceso distinto.
Veamos los predicados generados por DNF:

```
releaseWrite_dnf_1(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer=P &
  WriterLockState=locked.

releaseWrite_dnf_2(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=unlocked.

releaseWrite_dnf_3(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer neq P &
  WriterLockState=locked.
```

Podemos ver que en todos los casos los valores que toman *WriterLockState* y *Writer* están completamente condicionados por lo que no hay ninguna táctica apropiada que se pueda aplicar sobre estas variables de estado que generen casos satisfacibles distintos a los que ya se tienen.

Esto nos lleva a aplicar tácticas que involucren a las otras dos variables de estado, por un lado se podría aplicar Intervalo de Enteros a *MaxReaders*, pero esto no genera ningún caso de prueba interesante porque tener diferentes valores máximos no afecta de ninguna manera a esta operación. Por otro lado tenemos el conjunto de lectores *Readers* donde la única táctica factible a aplicar es Cardinalidad de Conjuntos, el problema que tiene esto es que al aplicarla sobre *releaseWrite_dnf_1* y *releaseWrite_dnf_3* genera casos de prueba donde la variable de estado *WriterLockState* es igual a *locked*, pero el conjunto de lectores es distinto de vacío, esto viola la invariante *InvNoReadersWhileWriter*. Esto nos deja solo con la rama *releaseWrite_dnf_2* donde

al aplicarle Cardinalidad de Conjuntos al conjunto de lectores estaríamos generando casos de prueba donde se está intentando soltar el lock en modo escritura, pero este no fue tomado por ningún proceso y la cantidad de lectores varía entre cero y dos para los distintos casos generados. Si bien estos casos de prueba generados son válidos, consideramos que no prueban alternativas funcionales importantes. De todas maneras se incluyó la aplicación de esta táctica.

Los comandos utilizados para generar los casos de prueba fueron:

```
ttf(rwlock).
applydnf(releaseWrite(P)).
applysc(releaseWrite_dnf_2,Readers).
prunett.
gentc.
exporttt.
```

El árbol de pruebas generado es el siguiente:

```
releaseWrite_vis
  releaseWrite_dnf_1 -> releaseWrite_tc_1
  releaseWrite_dnf_2
    releaseWrite_sc_21 -> releaseWrite_tc_21
    releaseWrite_sc_22 -> releaseWrite_tc_22
    releaseWrite_sc_23 -> releaseWrite_tc_23
  releaseWrite_dnf_3 -> releaseWrite_tc_3
```

Los casos de pruebas se guardaron en el archivo **rwlock_releaseWrite-tt.slog**:

```
releaseWrite_ts_1(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer=P &
  WriterLockState=locked.


releaseWrite_tc_1(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer=u(processPrevAxiom:n0) &
  P=u(processPrevAxiom:n0) &
  WriterLockState=locked.
```

```
releaseWrite_ts_21(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=unlocked &
  Readers={}.

releaseWrite_tc_21(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=unlocked &
  Readers={}.

releaseWrite_ts_22(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=unlocked &
  Readers={_59792} &
  dec(_59792,sum([creatorProcess,u(processPrevAxiom)])).

releaseWrite_tc_22(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=unlocked &
  Readers={u(processPrevAxiom:n0)}.

releaseWrite_ts_23(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=unlocked &
  Readers={_59792,_59862/_59864} &
  _59792 neq _59862 &
  dec([_59792,_59862],sum([creatorProcess,u(processPrevAxiom)])) &
  dec(_59864,set(sum([creatorProcess,u(processPrevAxiom)]))).

releaseWrite_tc_23(P,Readers,WriterLockState,Writer,MaxReaders) :-
  WriterLockState=unlocked &
  Readers={u(processPrevAxiom:n0),u(processPrevAxiom:n1)}.

releaseWrite_ts_3(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer neq P &
```

```
  WriterLockState=locked.


releaseWrite_tc_3(P,Readers,WriterLockState,Writer,MaxReaders) :-
  Writer=u(processPrevAxiom:n0) &
  P=u(processPrevAxiom:n1) &
  WriterLockState=locked.
```