- **Explique de qué manera el model checker establece si una dada propiedad es válida o no. Indique además porque interamente el model checker niega la propiedad a probar.**

El Model Checker es una técnica automatizada, que dado un modelo de sistema y una/s especificación de propiedad (definida en lógica temporal), trata de dar una verificación exhaustiva de correctitud, si la propiedad se verifica para el modelo.
En sí, el model checker trata de verificar si, la propiedad a cumplir, se cumple: si lo hace se fija si queda otra propiedad a verificar, si no la hay finaliza. En caso de que alguna propiedad no se cumpla responde con un o más contraejemplos.

Internamente el **model checker** trabaja con un autómata del modelo y otro para la propiedad (negada), y luego, trabaja con un autómata intersección de estos, ya que reconoce el lenguaje intersección (**Lint**).

$$\textbf{Lint} = L\omega(B_{TS}) \cap L\omega(B_{\neg p})$$

Demuestra (MC) que se cumple la propiedad **negandola** y buscando un contraejemplo, estilo prueba por el absurdo.

- Si **Lint** es vacío entonces no se encontró contraejemplo, por lo tanto se verifica la **P**. Significa que todas las ejecuciones del programa original pertenecen al lenguaje de **P**.
- Si **Lint** no es vacío entonces, encontré una propiedad que satisface **¬P**, por lo que la **P** original no vale, ya que hay, al menos, un ejemplo de ejecución donde **P** no vale.

Si no negará la propiedad al verificar (MC), tendría que verificar que la propiedad se cumpla en todas las ejecuciones posibles que pueden suceder en el programa.Esto se vuelve imposible, ya que el **Lint'** es, potencialmente, **infinito** y para probar que la propiedad vale, se debería probar que todas las cadenas que pertenecen al lenguaje están ahí, cosa que no es posible (problema de lenguajes recursivos).

- **Por qué resulta tan crítico asegurar la ausencia de efectos colaterales.**

Esto es así, ya que si sabemos que si un método tiene efectos colaterales entonces, estos, no se podrán capturar en una especificación. Ya que es **algo más** que está sucediendo en el cómputo, un cambio, y que **no está definido en el contrato establecido**. Y que, además, **tiene un efecto en el comportamiento de todo el servicio**. Y esto hace que **no** se pueda especificar formalmente, *por eso algunas veces las pruebas no cierran, porque hay algo más que está cambiando que no debería haber cambiado.*

- **Como se puede asegurar la ausencia de efectos colaterales e JML. Indique las diferencias.**

Hay dos formas para asegurar la ausencia de efectos colaterales en JML, una de ellas es la **cláusula assignable \nothing** y la otra es la cláusula **pure**.

- Se diferencia en que, pure es global al método, osea que lo ven todos y al establecer un contrato se sabe que ese método no va a tener efectos colaterales, y además prohíbe no-terminación y las excepciones.
- Mientras que **assignable \nothing** local a un caso de especificación (si ese servicio es utilizado en otro lugar, este, no sabe que el servicio es libre de efectos colaterales).

- **Qué opciones tiene para la verificación de ciclos repetitivos (provistas desde la especificación JML). Indique diferencias.**

Existen diferentes aproximaciones para probar la correctitud de un ciclo repetitivo.

- Para hacer una **correctitud total** de ciclos, se debe asegurar que el ciclo siempre termina, por lo que se utilizan las cláusulas **Loop invariant, decreasing** ‘x’. Donde x es una variable que decrece en cada ciclo hasta llegar a 0, con esto aseguramos que el el ciclo termina.
- Y para hacer una **correctitud parcia**l de ciclos, donde si termina el ciclo cumple lo que quería probar y/ó puede o no llegar a terminar, donde se utilizan las cláusulas **Loop invariant**, con **diverges true** (la no-terminación está permitida). ((Diverges false especifica la condición que debe satisfacerse antes de llamar a la operación si es que la operación no termina)).