Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba

Trabajo Final de Grado

Offbeat: Una extensión de PRISM para el análisis de sistemas temporizados tolerantes a fallas

Autor:
Nicolás Emilio Bordenabe.
Director:
Pedro R. D'Argenio.

28 de Marzo de 2011

A mi familia, que me apoyó de manera imprescindible en este trabajo,
y a Andrea, por acompañarme a lo largo de este trayecto.

Resumen

Los sistemas tolerantes a fallas son aquellos que son capaces de seguir operando luego de la ocurrencia de una o más fallas. Una falla puede provocar cambios no deseados en el estado interno del sistema, y para que el sistema tolere la falla, deberá ser capaz de soportar estos cambios y continuar operando de la manera esperada. Los sistemas tolerantes a fallas son comunes en casos donde una falla no es aceptable, ya que la misma puede derivar grandes pérdidas, tanto económicas como de vidas humanas. Otra técnica para garantizar que un sistema funcione de la manera que corresponde es el model checking, una técnica de verificación automática que permite determinar se el modelo de un sistema cumple una propiedad determinada. En caso de que el sistema no satisfaga la propiedad, el model checker generalmente proporciona un contraejemplo de ayuda para determinar la fuente del error. En el presente trabajo se detallan los modelos e ideas usadas para la construcción de un model checker probabilista temporizado, pensado para la verificación de sistemas tolerantes a fallas. Se describirá el proceso de inyección de fallas en los modelos formales, la sintaxis del lenguaje de la herramienta, el proceso de traducción del mismo al lenguaje de PRISM (el model checker sobre el cual se provee la capa de abstracción), y la aplicación de la herramienta desarrollada a dos casos de estudio.

Clasificación: D.2.4 - Software / Program Verification.

Palabras clave: model checking, sistema tolerante a fallas, verificación formal.

Agradecimientos

En esta sección me gustaría agradecer a muchas personas que me acompañaron y aportaron, de una forma u otra, al desarrollo de este trabajo. Mencionaré solo a algunos de ellos, pues son los que en este momento me vienen a la mente.

A mis padres, con quienes vivo, por brindarme con su sacrificio todo tipo de comodidades en pos de poder finalizar mis estudios.

A Andrea, que me acompaña prácticamente desde el inicio de mis estudios, por darme tanto amor y tanta fuerza en este gran viaje.

A mis hermanos y sus respectivas familias, que siempre me hicieron el aguante y estuvieron ahí cuando los necesité.

A Pedro, porque a pesar de ser la persona más atareada que conozco nunca dejó de ocuparse de revisar y corregir este trabajo.

A mis compañeros de la "Elite" y el "Dream Team", por tantos trabajos en grupo, laboratorios a último momento, cocas, metegoles en horario de clases, capitalistas y jodetes.

A mis compañeros de OMA y ACM, con quienes desde el secundario hasta hoy compartimos entrenamientos, películas, charlas geeks, y viajes inolvidables.

Índice general

Capítulo 1

Introducción

Actualmente, los sistemas informáticos están presentes en nuestra vida cotidiana en miles de formas diferentes. No solo en computadoras o teléfonos celulares, sinó en electrodomésticos, maquinaria industrial y vehículos, entre otros. El acceso a dichos sistemas se ha expandido, al punto de que hoy en día casi cualquier persona es capaz de interactuar con una computadora, un smartphone, un microondas o un sistema de audio.

Una *falla* es un suceso que atenta contra el funcionamiento normativo de un sistema, generalmente con resultados no deseados. Ante la ocurrencia de una falla, un sistema puede dejar de funcionar o comenzar a hacerlo de forma incorrecta. También es posible que una falla afecte el resultado final que debe ser calculado por un sistema. Las fallas pueden ser producto tanto de factores externos, como por ejemplo el mal funcionamiento de un componente o las condiciones climáticas, o de factores internos, es decir errores de programación o de diseño.

Si bien en algunos sistemas, tales como programas de software o controladores de electrodomésticos, la ocurrencia de una falla no tiene consecuencias mayores, en otros, tales como artefactos de uso médico o sistemas de control de vehículos, la necesidad de que sean confiables aumenta. En sistemas complejos, asegurar confiabilidad supone todo un reto. Y si bien algunos errores pueden ser causados por fuentes externas (código malicioso, desperfectos técnicos), en muchos casos las fallas son consecuencia de errores de diseño. Estos errores pueden traer como resultado efectos que van desde el reinicio de una aplicación de PC hasta la pérdida de vidas humanas[5]. Una falla, producto de la conversión de un número de punto flotante de 64 bits a un entero de 16 bits causó que el Ariane-5 explotara 36 segundos después del lanzamiento[23]. Un error en el sistema de manejo de equipaje pospuso la inauguración del aeropuerto de Denver por 9 meses, generando pérdidas de 1.1 millones de dólares por día[14, 25]. Y estos son solo algunos ejemplos.

Dado que gran parte de los errores proviene de la etapa de diseño del sistema, también es desable contar con técnicas que permitan la detección y

corrección temprana de errores que puedan resultar en futuras fallas. Actualmente hay dos técnicas comunmente usadas para este propósito: el *testing* y el *model checking*. La primera consiste en evaluar el comportamiento del sistema en un subconjunto del conjunto total de escenarios posibles para el mismo, mientras que la segunda hace referencia a una técnica de verificación automática de propiedades sobre un modelo del sistema. El testing puede demostrar la presencia de errores, pero no su ausencia[15].

El model checking es una técnica de verificación formal, es decir, una manera de probar formalmente que el sistema cumple una propiedad. Una de las ventajas del model checking es que es una técnica de verificación automática. El hecho de que no sea necesaria demasiada interacción con el usuario es visto como una ventaja en el sector industrial, ya que permite el uso de la herramienta por parte de usuarios no expertos[28] o con poca experiencia en ramas de la matemática.

El problema principal de las herramientas de model checking es la *explosión de estados*, que se produce como consecuencia de que el espacio de estados del modelo del sistema es exponencial al tamaño del sistema. Este problema suele ser el factor delimitante al momento de aplicar metodologías de verificación automática en sistemas de gran tamaño[24]. Uno de los métodos más usados para controlar el problema de la explosión de estados es el model checking *simbólico*. El mismo evita la construcción explícita de todo el grafo de estados, usando fórmulas booleanas para representar conjuntos de estados y relaciones. Estas fórmulas son representadas a través de una estructura de datos llamada *Diagramas de decisión binarios* (BDD), que escencialmente son grafos dirigidos acíclicos.

Esta técnica fue implementada en el model checker SMV[24], y uno de los logros más importantes obtenidos con ella[26] fue la verificación exitosa del protocolo de coherencia de caché del estandar **IEEE Futurebus**+[11].

En este trabajo presentaremos una herramienta de model checking, ideada para la verificación de sistemas tolerantes a fallas. La misma proporciona una capa de abstracción sobre PRISM, un model checker simbólico con soporte para modelos probabilísticos y temporizados. Nuestra herramienta proporciona un lenguaje similar al de PRISM para describir los modelos, junto con algunas construcciones agregadas para declarar fallas, especificando su probabilidad de ocurrencia, sus efectos y condiciones de rstauración, entre otros parámetros. La idea central es lograr separar la declaración de las fallas con de la especificación de los módulos, de manera de hacer el código más claro y reusable. El trabajo aquí desarrollado es una extensión del trabajo realizado por Edgardo Hames[16], en el cual se presenta una herramienta similar basada en el model checker NuSMV. El principal aporte de este trabajo será agregar una noción de comportamiento probabilista y temporal de las fallas (y de los modelos en general), que permita la verificación de modelos más complejos y fieles al comportamiento real del sistema.

Disposición de este trabajo

Éste trabajo está organizado de la siguiente manera. En el Capítulo 2 se dará una explicación de los modelos y lógicas usadas en el model checking probabilista temporizado. Se explicará además, la técnica del model checking y sus pasos, y se presentará el model checker PRISM sobre el cual trabaja nuestra herramienta. El Capítulo 3 dará una breve explicación de los sistemas tolerantes a fallas, y mostrará el proceso de model checking para este tipo de sistemas. El Capítulo 4 dará una definición formal del concepto de falla y de su relación con los modelos usados para describir los sistemas. También definirá formalmente un modelo probabilístico con fallas, estableciendo su semántica en términos de los modelos descriptos en el Capítulo 2. En el Capítulo 5 se describirá a OFFBEAT, su sintaxis, opciones, y como se efectúa la traducción de los modelos descriptos en el lenguaje de la herramienta a modelos PRISM. El Capítulo 6 presentará 2 casos de estudio reales en los cuales se realizan modelos y verificaciones usando la herramienta desarrollada, de manera de evaluar su uso y medir su desempeño. Finalmente, en el Capítulo 7 se da una conclusión del trabajo, dando un análisis de los resultados obtenidos, comentando los problemas y limitaciones que surgieron durante el desarrollo de la herramienta, y resaltando las posibilidades de trabajo a futuro.

Capítulo 2

Model Checking Probabilista Temporizado

El *model checking* es una técnica que permite verificar automáticamente si el modelo de un sistema cumple o no una propiedad especificada en alguna lógica temporal. Ejemplos de éstas propiedades pueden ser: *"El sistema nunca llega a un estado de deadlock"*, o *"Si un proceso está contínuamente esperando para entrar en su región crítica, eventualmente lo hará"*. Como se puede ver, a diferencia de las lógicas proposicionales, las lógicas temporales permiten razonar sobre comportamiento que puede variar a lo largo del tiempo.

Las propiedades se verifican sobre un *sistema de transiciones etiquetado*, el cual representa un modelo del sistema, y es generado por el model checker a partir de un modelo hecho por el usuario de la herramienta, en un lenguaje que la misma sepa procesar.

Los modelos a verificar pueden exhibir distintos tipos de comportamiento: no-determinista, probabilista, sujeto a la variación del tiempo, o alguna combinación de los anteriores. En cada caso, es necesario contar con un model checker que sepa entender ese tipo de modelo.

OFFBEAT permite la verificación de modelos no-deterministas probabilistas temporizados. A lo largo de este capítulo explicaremos brevemente la teoría detras de los formalismos y modelos usados, y describiremos a **PRISM**[1] [26], el model checker sobre el cual trabaja la herramienta. Para la descripción de los modelos, usaremos en su mayoría la notación y definiciones usadas en [30].

2.1. Conceptos Iniciales

En esta sección definiremos de manera gradual algunos conceptos que serán de suma utilidad para comprender las secciones subsiguientes.

[1] http://www.prismmodelchecker.org/

2.1.1. Sistemas de Transiciones Etiquetados

Un *sistema de transiciones etiquetado* (LTS) es una forma de modelar el comportamiento de un sistema, que puede estar compuesto por uno o más módulos que capaces de interactuar de manera sincrónica. Un LTS es similar a un grafo dirigido, cuyos nodos representan estados del sistema, y cuyas aristas representan acciones que pueden llevar al sistema de un estado a otro. Los estados describen información del sistema en un determinado momento; las transiciones, la forma en la que el sistema pasa de un estado a otro. Por ejemplo, en el caso de un programa secuencial de computadora, el estado puede representar el valor de las variables en un momento dado, y las transiciones, la ejecución de una sentencia; en el caso de un ascensor, el estado puede indicar el piso en el cual se encuentra el mismo, el estado de la puerta (abierta o cerrada), y si hay o no un exceso de peso, y las transiciones pueden indicar el movimiento de un piso a otro, la acción de cerrar o abrir la puerta, y el hecho de que suba o baje una persona.

La descripción de un LTS consta además de un conjunto finito de *proposiciones atómicas* (AP), que sirven como indicadores de algunas propiedades y características relevantes de los estados del sistema (por ejemplo, "*deadlock*", "$x - y \leq 10$").

Formalmente, un LTS es una estructura $\langle S, A, \rightarrow, L \rangle$, donde S es el conjunto de estados, A el conjunto de acciones (etiquetas), $\rightarrow \subseteq S \times A \times S$ es la relación de transición y $L : S \rightarrow 2^{AP}$ asigna el conjunto de proposiciones atómicas válidas en cada estado.

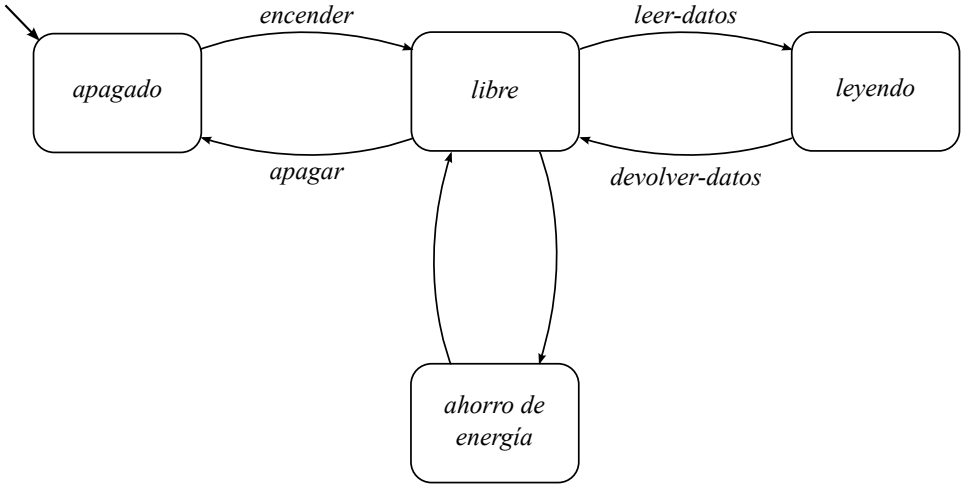


Figura 2.1: El LTS del dispositivo de lectura

Por ejemplo, el LTS de la figura 2.1 representa el comportamiento de un dispositivo de lectura, que comienza estando apagado. Una vez encendido, el dispositivo se encuentra libre hasta que se le solicita la lectura de datos, luego de la cual, devuelve los datos leidos y vuelve a estar libre. Si el dispositivo

está libre por mucho tiempo, entra en modo de ahorro de energía. En este ejemplo, AP = {apagado, libre, leyendo, ahorro de energía}.

Un *sistema de transiciones etiquetado temporizado* (LTTS) es una variación de un LTS, donde las transiciones, además de representar acciones realizadas por el sistema (transiciones discretas), pueden representar el paso del tiempo (transiciones de tiempo), y en este último caso, la "etiqueta" de la transición es la cantidad de tiempo que transcurre durante la misma. Es decir, un LTTS es un LTS de la forma $\langle S, A \cup \mathbb{R}_{\geq 0}, \rightarrow, L \rangle$.

Un *camino* en un LTTS es una sucesión infinita transiciones, que representan un posible comportamiento del sistema. La duración hasta el n-ésimo estado del camino es el tiempo que transcurre durante las primeras $n-1$ transiciones del camino (en las transiciones discretas, el tiempo que transcurre es 0). Un camino es divergente si y solo si, para cualquier $t \in \mathbb{R}^+$, existe un $i \in \mathbb{N}$ tal que la duración hasta el i-ésimo estado del camino es mayor que t. Un LTTS es *no-zeno* si y solo si, para cada estado alcanzable s existe al menos un camino divergente que comienza en el estado s.

2.1.2. Autómatas Teporizados

Para modelar el comportamiento de sistemas en donde el paso del tiempo es relevante, podemos usar *autómatas temporizados* (TA)[30, 5]. Este modelo es simplemente un autómata extendido con un conjunto finito de relojes, cuyos valores se van incrementando al mismo tiempo. Estos relojes son usados en guardas para algunas de las aristas, y en invariantes para algunos de los nodos. Además, los algunos relojes pueden ser reseteados al tomar algunas de las aristas.

La semántica de una autómata temporizado TA es un LTTS, que denominaremos $LTTS_{TA}$, y cuyos estados representan un estado del sistema en un momento determinado.

TA es *no-zeno* si y solo si $LTTS_{TA}$ es *no-zeno*.

2.1.3. Procesos de Decisión de Markov

Otra forma de modelar el comportamiento de un sistema es a través de un *proceso de decisión de Markov* (MDP)[26, 30], que permite expresar tanto comportamientos probabilistas como no-deterministas.

Se puede entender a un MDP como un conjunto finito de estados S, y un conjunto finito no vacío de distribuciones de probabilidad sobre S asociado a cada estado. Cada una de estas distribuciones μ asigna una probabilidad a cada uno de los estados de S, de manera tal que:

$$\sum_{s \in S} \mu(s) = 1$$

Para que el sistema, estando en un estado s, pueda pasar a otro estado s',

debe primero elejir no-determinísticamente una distribución μ asociada a s; luego, la probabilidad de que el sistema pase de s a s' es $\mu(s')$.

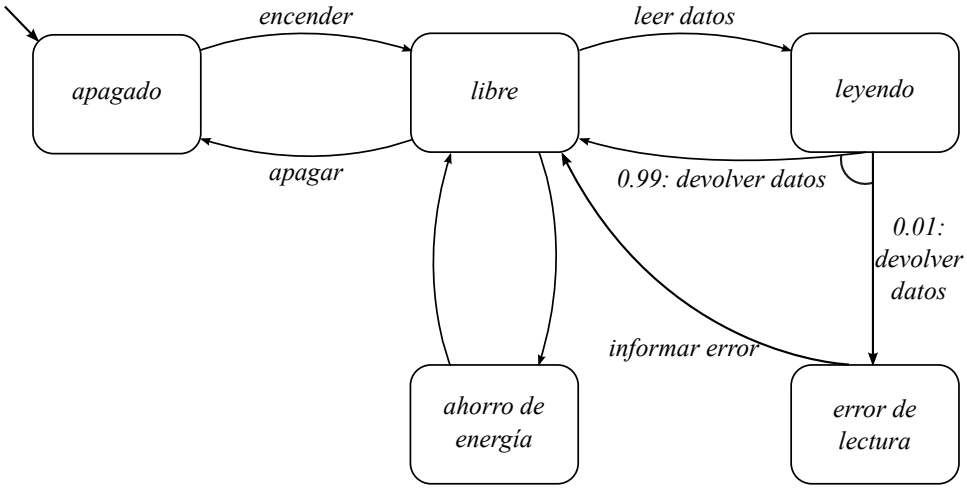


Figura 2.2: Un MDP del dispositivo de lectura presentado anteriormente

Las elecciones no-deterministas son hechas por un adversario o scheduler, que elije una opción en base a las elecciones hechas anteriormente. En este tipo de modelo, no tiene sentido hablar de propiedades como *"la probabilidad de que el sistema alcance un estado s"*, dado que la misma puede variar para distintos adversarios. Sin embargo, podemos hablar de probabilidades máximas y mínimas sobre el conjunto de todos los adversarios.

En la figura 2.2 podemos ver el ejemplo de dispositivo de lectura que presentamos en la sección 2.1.1, con algunas modificaciones. En este caso, el proceso de lectura tiene dos resultados posibles: es correcto con probabilidad 0.99 y resulta en un error de lectura con probabilidad 0.01.

2.1.4. Lógicas Temporales: CTL, PCTL y TCTL

Las lógicas temporales son aquellas que permiten tomar en cuenta distintos momentos en el tiempo para decidir si una sentencia es verdadera o falsa.

CTL (Computation Tree Logic) modela el tiempo en una estructura de árbol, donde cada camino representa un posible escenario del futuro[5]. Si hablamos de procesos, cada camino puede interpretarse como una posible ejecución del mismo. La sintaxis de CTL utiliza los mismos operadores de la lógica proposicional ($\wedge$, $\vee$, $\neg$, $\rightarrow$) junto con algunos nuevos:

- **X** ϕ, en el próximo estado vale ϕ.
- **F** ϕ, en algún momento en el futuro, vale ϕ
- **G** ϕ, siempre en el futuro vale ϕ

- ϕ **U** ψ, ϕ vale continuamente hasta que vale ψ

CTL distingue dos tipos de fórmulas: las fórmulas de estado, que son las clásicas fórmulas que podemos armar con los operadores de la lógica proposicional, y las fórmulas de camino, que usan los operadores temporales. Estos tipos de fórmula son evaluados sobre estados y caminos, respectivamente. Las propiedades de los modelos son siempre expresadas a través de fórmulas de estado. Las fórmulas de caminos solo pueden ocurrir *adentro* de las fórmulas de estado, precedidas por $\forall$ o $\exists$. Por ejemplo, la fórmula $\forall$ **F** ϕ es verdadera en un estado s si y solo si para todo camino que comience en s se cumple que, en algún momento en el futuro el sistema alcanza un estado en el cual se cumple ϕ.

PCTL (Probabilistic Computation Tree Logic)[26] es una extensión de CTL, que agrega a las fórmulas de estado el operador $\mathcal{P}_{\bowtie p}[\psi]$, donde ψ es una fórmula de camino y $\bowtie$ es uno de los siguientes: $<$, $\leq$, $>$, $\geq$, $=$. Intuitivamente, un estado s satisface la fórmula $\mathcal{P}_{\bowtie p}[\psi]$ si la probabilidad de tomar desde s un camino que satisfaga ψ está dentro del intervalo especificado por $\bowtie p$. En el caso de los MDP, la semántica del operador $\mathcal{P}_{\bowtie p}$ varía un poco, ya que la probabilidad debe estar dentro del rango $\bowtie p$ para todos los adversarios en consideración. En PCTL, a diferencia de CTL, las fórmulas de caminos solo ocurren como parámetro del operador $\mathcal{P}_{\bowtie p}$.

TCTL (Timed Computation Tree Logic)[30, 5] es otra extensión de CTL, que agrega a las fórmulas de estado los elementos de $Zonas(X)$, con X un conjunto de relojes. Además, agrega las extensiones temporizadas de los operadores de caminos:

- $\mathbf{F}^J$ ϕ, en algún instante $t' \in J$, vale ϕ
- $\mathbf{G}^J$ ϕ,para todo instante $t' \in J$, vale ϕ
- ϕ $\mathbf{U}^J$ ψ, $\phi \vee \psi$ vale continuamente hasta que, en algún instante de tiempo $t' \in J$ vale ψ

Donde $J \subseteq \mathbb{R}^+$. Se pueden ver a estos operadores como la versión generalizada de los operadores definidos en CTL, ya que, por ejemplo,

$$\phi \ \mathbf{U} \ \psi = \phi \ \mathbf{U}^{[0,\infty)} \ \psi$$

Sin embargo, cabe señalar que la semántica de ϕ $\mathbf{U}^J$ ψ permite que $\phi \vee \psi$ valga antes de que se cumpla ψ, a diferencia de la semántica de CTL, donde solo puede valer ϕ antes de ψ. La razón de ésto es técnica, debido a la naturaleza continua del tiempo, y está explicada en [5], aunque se puede ver que en CTL, ϕ **U** ψ es equivalente a $(\phi \vee \psi)$ **U** ψ.

Notar que en TCTL el operador **X** pierde sentido, pues dado que el tiempo es contínuo, no se puede hablar de un único "instante siguiente" en el tiempo.

2.2. Sistemas Etiquetados Probabilísticos Temporizados

Un *sistema etiquetado probabilístico temporizado*[30] (LTPS) puede verse como una extensión tanto de un proceso de decisión de Markov como de un sistema de transiciones etiquetado temporizado. Representa un sistema con una cantidad posiblemente infinita de estados, donde cada estado refleja la situación discreta y temporal del mismo. Las transiciones pueden representar tanto la ejecución de acciones por parte del sistema, como también el paso del tiempo.

Formalmente, un LTPS es una 5-upla $\langle S, \overline{s}, L, Act, Steps \rangle$, donde:

- S es el conjunto (posiblemente infinito) de estados,
- $\overline{s} \in S$ es el estado *inicial*,
- $L : S \to 2^{AP}$ es una función de etiquetas,
- Act es el conjunto de acciones del sistema, y
- $Steps \subseteq S \times (Act \cup \mathbb{R}^+) \times Dist(S)$ es la relación de transición probabilística, tal que si $(s, t, \mu) \in Steps$ para cualquier $t \in \mathbb{R}^+$, entonces μ es una *distribución punto* (es decir, exite $s \in S$ tal que $\mu(s) = 1$).

2.2.1. Caminos

Un *camino* en un LTPS es una secuencia infinita de transiciones:

$$\omega = s_0 \xrightarrow{t_0,\mu_0} s_1 \xrightarrow{t_1,\mu_1} s_2 \xrightarrow{t_2,\mu_2} \dots$$

donde, para todo i, $t_i \in Act$ implica $\mu_i(s_{i+1}) > 0$, y $t_i \in \mathbb{R}^+$ implica $\mu_i(s_{i+1}) = 1$. Los caminos representan posibles ejecuciones del sistema modelado. La duración hasta el $(n+1)$-ésimo estado de ω es

$$\mathcal{D}_\omega(n+1) = \sum_{i=0}^{n} d(t_i)$$

donde

$$d(t) = \begin{cases} t & t \in \mathbb{R}^+ \\ 0 & t \in Act \end{cases}$$

Al igual que con los sistemas de transiciones etiquetados temporizados, estamos interesados en modelar solo el comportamiento "realista" del sistema. Por lo tanto, solo debemos considerar los caminos que continuamente avanzan en el tiempo, y no aquellos que realizan un número infinito de acciones en una cantidad finita de tiempo. Decimos entonces que un camino ω es *divergente* si, para cualquier $t \in \mathbb{R}^+$, existe $i \in \mathbb{N}$ tal que $\mathcal{D}_\omega(i) > t$.

2.2.2. Adversarios

Un *adversario* de un LTPS es una forma de resolver las elecciones no-deterministas del sistema, basado en las elecciones hechas históricamente. Formalmente, sea $Caminos_{fin}$ el conjunto de todos los tramos iniciales de caminos posibles en un LTPS (es decir, los caminos "finitos" del LTPS), un adversario A es una función $A : Caminos_{fin} \to Steps$, tal que

$$A(s_0 \xrightarrow{t_0,\mu_0} \ldots \xrightarrow{t_{n-1},\mu_{n-1}} s_n) = (s_n, t_n, \mu_n), \text{ con } (s_n, t_n, \mu_n) \in Steps$$

Podemos definir entonces el conjunto $Caminos^A(s)$ como el conjunto de todos los caminos que comienzan en s y que son posibles bajo el adversario A. También definiremos la función de probabilidad

$$Prob_s^A : Caminos^A(s) \to \mathbb{R}^+$$

tal que $Prob_s^A(\omega) = \delta$ si y solo si la probabilidad de tomar desde s el camino ω bajo el adversario A es igual a δ.

Diremos que un adversario A es *divergente* si, para todo estado s del LTPS, la probabilidad sobre A de tomar un camino no-divergente desde s es 0. Formalmente, dado un sistema etiquetado probabilístico temporizado $LTPS$, un adversario A de $LTPS$ es divergente si, para todo estado s de $LPTS$, se cumple

$$\sum_{\omega \in Caminos^A(s)} Prob_s^A(\omega) = 1$$

Llamaremos Adv_{LTPS} al conjunto de adversarios divergentes de $LTPS$. Como dijimos antes, solo nos interesan los modelos que reflejen un comportamiento realista del sistema. Por lo tanto, exigiremos que los LTPS tengan al menos un adversario divergente. Diremos que un LTPS que cumple esta condición es *no-zeno*.

2.3. Relojes y Zonas

En ésta sección usaremos la misma notación utilizada en [30]. Llamaremos *relojes* a un conjunto de variables sobre $\mathbb{R}^+$. Estas variables servirán para modelar el paso del tiempo. Los relojes incrementan su valor implícitamente a medida que el tiempo transcurre, y pueden ser reiniciados a 0. A diferencia de otros tipos de variables, las únicas operaciones válidas sobre un reloj son la de consultar su valor y la de reiniciar su valor a cero. Es decir, el valor de un reloj representa la cantidad de tiempo transcurrido desde la última vez en que el mismo ha sido reiniciado[5].

Dado un conjunto de relojes $\mathcal{X} = \{x_1, .., x_n\}$, una valuación es una función $v : \mathcal{X} \to \mathbb{R}^+$ que asigna a cada reloj $x \in \mathcal{X}$ un valor real no negativo

$v(x)$. Dado un conjunto $\mathcal{Z} \subseteq \mathcal{X}$, la valuación $v[\mathcal{Z} := 0]$ es aquella que asigna a cada reloj de $\mathcal{Z}$ el valor 0, y a los demás el valor correspondiente a v.

El conjunto de *zonas* de $\mathcal{X}$, que denotaremos $Zonas(\mathcal{X})$, está definido inductivamente de la siguiente forma:

$$n ::= x < c \mid x \leq c \mid x - y < c \mid x - y \leq c \mid \neg n \mid n \vee n$$

donde $x, y \in \mathcal{X}$ y $c \in \mathbb{Z}$. Una zona es *atómica* cuando es de la forma $x < c$, $x \leq c$, $x - y < c$, ó $x - y \leq c$.

Una valuación satisface una zona cuando el valor de verdad de la misma al reemplazar cada reloj por el valor que le asigna la valuación, es *true*. Gracias a esto, podemos interpretar la semántica de una zona como el subconjunto de valuaciones que hacen que el valor de verdad de la misma sea *true*. Sea v una valuación y g una zona, escribiremos $v \lhd g$ si v satisface g.

2.4. Autómatas Probabilistas Temporizados

El tipo de modelo que usaremos son los *autómatas probabilistas temporizados* (PTA), que sirven para describir sistemas que exhiben comportamiento no-determinista, estocástico, y sujeto a los cambios en el tiempo. Como la mayoría de los modelos, puede ser usado para describir o bien un solo sistema probabilista temporizado, o bien varios de éstos interactuando de manera sincrónica o asincrónica.

Los PTA son una extensión tanto de los MDP, como de los TA[30]. Formalmente, dado un conjunto fijo finito de proposiciones atómicas AP, un PTA es una 7-upla $\langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob \rangle$ tal que:

- L es un conjunto finito de nodos,
- $\mathcal{L} : L \to 2^{AP}$ es una función que asigna a cada nodo del autómata el conjunto de proposiciones atómicas que son verdaderas en ese nodo,
- $\bar{l} \in L$ es el nodo inicial,
- Act es un conjunto finito de etiquetas de acciones,
- $\mathcal{X}$ es un conjunto finito de relojes,
- $inv : L \to Zonas(\mathcal{X})$ es una función que asigna a cada nodo un invariante, y
- $prob \subseteq L \times Zonas(\mathcal{X}) \times Act \times Dist(2^{\mathcal{X}} \times L)$ es una relación de transición probabilística, que consiste en un nodo de origen, una condición de habilitación, una etiqueta de acción, y un conjunto de distribuciones de probabilidad que asignan probabilidades a pares de la forma (X, l') para un conjunto X de relojes a resetear, y un nodo destino l'.

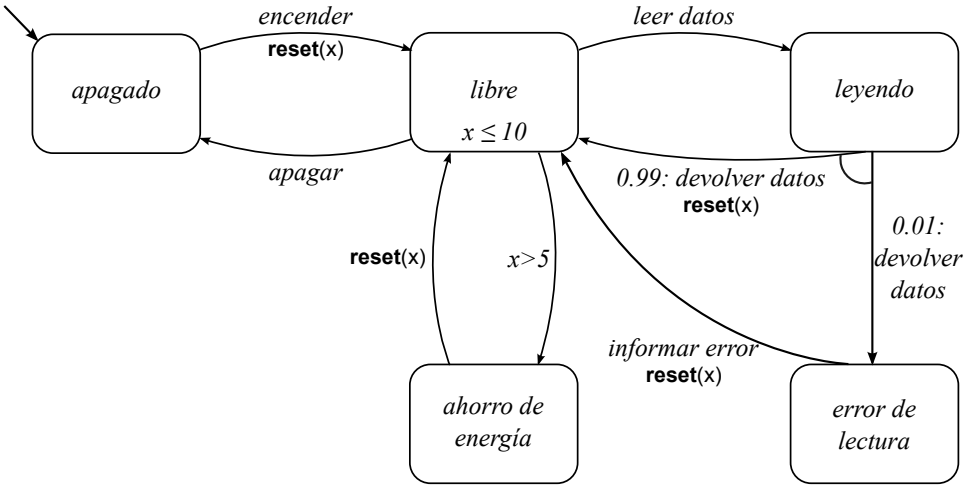


Figura 2.3: Un PTA del dispositivo de lectura presentado anteriormente

Los nodos del PTA representan los posibles estados del sistema. Cada nodo l tiene asociados un conjunto de proposiciones atómicas $\mathcal{L}(l)$ que son ciertas en ese estado del sistema, y un invariante $inv(l)$ que representa las restricciones de tiempo asociadas a ese estado. Intuitivamente, el sistema solo puede permanecer en el estado representado por l mientras $inv(l)$ sea verdadero. Además, l tiene un conjunto de aristas de salida, que representan un posible cambio de estado del sistema. Dada una transición $(l, g, a, p) \in prob$, el sistema puede ejecutarla si se encuentra en el nodo l y la guarda g es verdadera en la valuación actual. En tal caso, se ejecuta la acción a y, con probabilidad $p(X, l')$, se resetean todos los relojes en X y se pasa al nodo l'.

En el PTA de la figura 2.3, al dispositivo de lectura le agregamos información sobre el paso del tiempo, usando un reloj x. El dispositivo no puede permanecer en estado libre por más de 10 minutos, y a partir de los 5 minutos en esa condición, puede pasar a modo de ahorro de energía. Notar que es necesario resetar x cada vez que pasamos al estado *libre*.

2.4.1. Semántica de un PTA

La semántica de un autómata probabilista temporizado se define en términos de un sistema etiquetado probabilista temporizado.

Sea $PTA = \langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob \rangle$ un autómata probabilista temporizado, la *semántica* de PTA está definida como el sistema etiquetado probabilista temporizado $LTPS_{PTA} = \langle S, \bar{s}, \mathcal{L}', Act, Steps \rangle$ donde:

- $S \subseteq L \times \mathbb{R}^+$ y $(l, v) \in S$ si y solo si $v \lhd inv(l)$,
- $((l, v), t, \mu) \in Steps$ si y solo si se cumple una de las siguientes condiciones:

- Transiciones de tiempo: $t \geq 0$, $\mu(l, v+t) = 1$ y $v + t' \lhd inv(l)$ para todo $0 \leq t' \leq t$, donde $v + t'$ representa la valuación que asigna a cada reloj x el valor $v(x) + t'$.
- Transiciones discretas: $t = 0$ y existe $(l, g, p) \in S$ tal que:
  - $v \lhd g$,
  - para todo (X, l'), $p(X, l') > 0$ implica $(l', v[X := 0]) \in S$, y
  - para todo $(l', v') \in S$:

$$\mu(l', v') = \sum_{X \subseteq \mathcal{X} \ \&\ v' = v[X:=0]} p(X, l')$$

- $\mathcal{L}'(l, v) = \mathcal{L}(l)$ para todo $(l, v) \in S$.

Decimos que PTA es no-zeno si y solo si $LTPS_{PTA}$ es no-zeno.

El sistema, representado por $LTPS_{PTA}$ comienza su ejecución con todos los relojes seteados a 0. El valor de los mismos aumenta a la misma velocidad. El sistema puede permanecer "quieto" en un estado siempre y cuando el invariante correspondiente se satisfaga (en el LTPS, esto se representa mediante una o más transiciones de tiempo consecutivas). Si la guarda de una transición es satisfecha, el sistema puede entonces realizar una transición discreta. La misma es instantánea, es decir, el valor de los relojes no aumenta cuando esta se toma. La elección de la transición discreta es tanto no-determinista como probabilista: Una vez que se escoge no-determinísticamente la transición habilitada a tomar, el sistema ejecuta el paso probabilístico, de acuerdo a la distribución asociada a la transición elegida. Adicionalmente, un subconjunto de los relojes del sistema pueden ser reseteados a 0 al ejecutar dicha transición.

2.4.2. Composición Paralela

Al momento de definir sistemas complejos, es útil contar con la *composición paralela* de dos o más autómatas probabilistas temporizados.

Sean PTA_1 y PTA_2 tal que $PTA_i = \langle L_i, \mathcal{L}_i, \bar{l}_i, Act_i, \mathcal{X}_i, inv_i, prob_i \rangle$, con $i = 1, 2$, donde $\mathcal{X}_1 \cap \mathcal{X}_2 = \emptyset$. La *composición paralela* de PTA_1 y PTA_2 es el autómata probabilista temporizado

$$PTA_1 \parallel PTA_2 = \langle L_1 \times L_2, \mathcal{L}_1 \cup \mathcal{L}_2, (\bar{l}_1, \bar{l}_2), Act_1 \cup Act_2, \mathcal{X}_1 \cup \mathcal{X}_2, inv, prob \rangle,$$

donde $inv(l, l') = inv_1(l) \wedge inv_2(l')$ para todo $(l, l') \in L_1 \times L_2$, y $((l_1, l_2), g, \sigma, p) \in prob$ si y solo si se cumple alguna de las siguientes condiciones:

1. $\sigma \in Act_1 \setminus Act_2$ y existe $(l_1, g_1, \sigma_1, p_1) \in prob_1$ tal que $p = p_1 \otimes \chi_{l_2}$, $g = g_1$ y $\sigma = \sigma_1$.
2. $\sigma \in Act_2 \setminus Act_1$ y existe $(l_2, g_2, \sigma_2, p_2) \in prob_2$ tal que $p = \chi_{l_1} \otimes p_2$ y $g = g_2$ y $\sigma = \sigma_2$.

3. $\sigma \in Act_1 \cap Act_2$ y existe $(l_1, g_1, \sigma_1, p_1) \in prob_1$ y $(l_2, g_2, \sigma_2, p_2) \in prob_2$ tal que $p = p_1 \otimes p_2$ y $g = g_1 \wedge g_2$.

donde $p_1 \otimes p_2(X_1 \cup X_2, (l_1, l_2)) = p_1(X_1, l_1) \cdot p_2(X_2, l_2)$, con $l_1 \in L_1$, $X_1 \in \mathcal{X}_1$, $l_2 \in L_2$ y $X_2 \in \mathcal{X}_2$, y χ_l es la función característica del nodo l, tal que $\chi_l(\emptyset, l) = 1$.

La condición 3 establece que, para que dos autómatas puedan sincronizar al realizar una acción, ésta deberá aparecer con el mismo nombre en ambos y la guarda de las dos debe ser satisfecha.

2.5. PTCTL

Para expresar las propiedades que queremos verificar en nuestro sistema, usaremos la lógica probabilística en tiempo real, PTCTL (Probabilistic Timed Computation Tree Logic), que surge como una combinación de dos de las lógicas descriptas en la sección 2.1.4: PCTL y TCTL. Con la misma se pueden expresar fórmulas sobre autómatas probabilistas temporizados, es decir, sobre nuestros modelos de sistemas, y de esta forma verificar en los mismos propiedades que se refieran tanto al comportamiento estocástico del sistema como al comportamiento en relación al paso del tiempo. Por ejemplo, una propiedad válida en PTCTL es *"La probabilidad de que el sistema llegue a un estado estable en a lo sumo 10 segundos es mayor o igual a 0.98"*.

La definición de propiedades en PTCTL es muy parecida a la de TCTL, pero agregando el operador $\mathcal{P}_{\bowtie p}$. La sintaxis de PTCTL se define formalmente de esta manera:

$$\phi ::= a \mid g \mid \neg\phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}[\psi]$$

$$\psi ::= \phi \ \mathbf{U}^J \ \phi \mid \mathbf{F}^J \ \phi \mid \mathbf{G}^J \ \phi$$

Con $a \in AP$, $g \in Zonas(\mathcal{X})$, $J \subseteq \mathbb{R}^+$, $\bowtie \in \{\leq, <, >, \geq\}$, y donde ϕ representa fórmulas de estado y ψ representa fórmulas de camino.

Dado un autómata probabilista temporizado $PTA = \langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob \rangle$, las fórmulas de estados y las fórmulas de caminos se verifican sobre los estados y caminos de $LTPS_{PTA}$, respectivamente.

Sean (l, v) y $\omega = (l_0, v_0) \xrightarrow{t_0,\mu_0} (l_1, v_1) \xrightarrow{t_1,\mu_1} (l_2, v_2) \xrightarrow{t_2,\mu_2} \ldots$ un estado y un camino de $LTPS_{PTA}$ respectivamente. La relación de satisfacción $\models$ está definida de la siguiente manera:

$$(l,v) \models true$$

$$(l,v) \models a \quad \Leftrightarrow \quad a \in \mathcal{L}(l)$$

$$(l,v) \models g \quad \Leftrightarrow \quad v \triangleleft g$$

$$(l,v) \models \neg\phi \quad \Leftrightarrow \quad (l,v) \not\models \phi$$

$$(l,v) \models \phi_1 \wedge \phi_2 \quad \Leftrightarrow \quad (l,v) \models \phi_1 \text{ y } (l,v) \models \phi_2$$

$$(l,v) \models \mathcal{P}_{\bowtie p}[\psi] \quad \Leftrightarrow \quad p^A_{(l,v)}(\psi) \bowtie p \text{ para todo } A \in Adv_{LTPS_{PTA}}$$

donde, para todo adversario $A \in Adv_{LTPS_{PTA}}$, s un estado de un LTPS, y ψ una fórmula de camino:

$$p^A_s(\psi) = Prob^A_s(\{\omega \in Caminos^A(s) \mid \omega \models \psi\})$$

Finalmente,

$$\omega \models \phi_1 \ \mathbf{U}^J \ \phi_2 \quad \Leftrightarrow \quad \text{existen } i \in \mathbb{N} \text{ y } t \in [0, d(t_i)] \text{ tales que}$$

$$\mathcal{D}_\omega(i) + t \in J,$$

$$(l_i, v_i + t) \models \phi_2,$$

$$\text{para todo } j < i \text{ y } t' \in [0, d(t_j)] \text{ , tenemos}$$

$$(l_j, v_j + t') \models \phi_1 \vee \phi_2$$

$$\text{para todo } t' \in [0, t), \text{ tenemos}$$

$$(l_i, v_i + t') \models \phi_1 \vee \phi_2$$

No es necesario dar la semántica de los operadores $\mathbf{F}^J$ y $\mathbf{G}^J$, ya que ambos son casos particulares del operador $\mathbf{U}^J$:

$$\mathbf{F}^J \ \phi = true \ \mathbf{U}^J \phi$$

$$\mathbf{G}^J \ \phi = \neg(\mathbf{F}^J \ \neg\phi)$$

2.6. Model Checking

El model checking es una técnica de verificación automática que, dados un modelo de un sistema y una propiedad a verificar, determina si el modelo cumple o no con la propiedad, y en caso de que no lo haga, por lo general proporciona un contraejemplo que ayuda a localizar el origen del error. Una

de los principales problemas de los model checkers es el fenómendo de la *explosión de estados*, que hace referencia a que la verificación de sistemas muy complejos o con muchas componentes interactuando entre si genera modelos con una cantitad muy grande de estados que superan con facilidad la capacidad de procesamiento de cualquier computadora ordinaria. Una de las técnicas más efectivas para disminuír la explosión de estados es el model checking *simbólico*, donde se usan formulas booleanas para representar conjuntos de estados.

2.6.1. El proceso del Model Checking

Dado un sistema y un requerimiento, el proceso del model checking puede dividirse en 3 grandes pasos[12, 16]:

Modelado. En esta etapa se describe el comportamiento relevante del sistema en un lenguaje de modelado que el model checker sepa entender. A veces es necesario abstraer algunos detalles del sistema, o incluso alguna funcionalidad del mismo en base al requerimiento que se desee verificar, con el objetivo de evitar la explosión de estados.

Especificación. En este paso se expresa el requerimiento como una fórmula en algun formalismo, generalmente una lógica temporal, y se escribe en un lenguaje que el model checker comprenda.

Verificación. En la mayoría de los model checkers, este paso involucra el proceso *automático* de verificación de la propiedad sobre el modelo de nuestro sistema. Sin embargo, lo normal es que el usuario deba intervenir, no solo para interpretar los resultados de la verificación, sino también para ajustar algunas de las opciones de la herramienta y realizar correcciones al modelo.

En la figura 2.4 puede verse una versión gráfica del proceso. Inicialmente tenemos un sistema y un requerimiento, y queremos verificar si este requerimiento es satisfecho por el sistema. Por lo tanto, construimos un modelo del sistema y especificamos el requerimiento como una propiedad lógica. Entonces, podemos proceder a la etapa de verificación, en la cual el model checker decide si el modelo cumple la propiedad. En caso de que no lo haga, generalmente, el model checker proporciona un contraejemplo o *traza*, que muestra un comportamiento del sistema en el cual la propiedad no es satisfecha y que sirve de ayuda para depurar el error. Sin embargo, en algunos casos este contraejemplo puede ser bastante complicado de expresar como traza. Esto sucede principalmente en los model checkers probabilistas, donde las trazas para propiedades que implican cuantificación de probabilidades son particularmente difíciles de expresar, y por lo general suelen darse como expresiones regulares. Algunos metodologías para construir este tipo de contraejemplos pueden encontrarse en [13, 3].

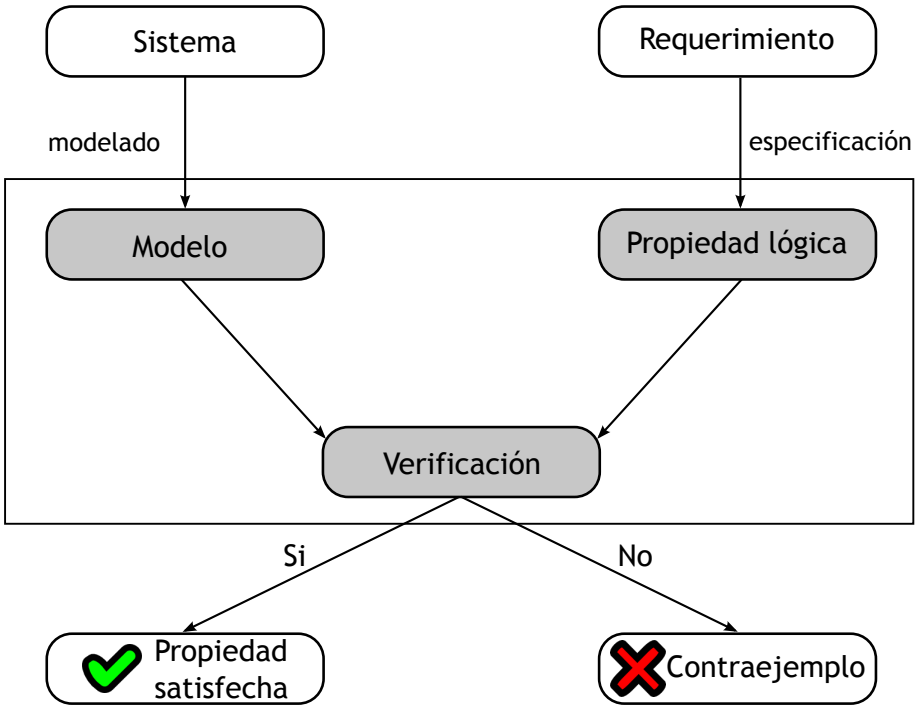


Figura 2.4: El proceso clásico de model checking[16]

Las secciones dentro del recuadro en la figura 2.4 representan los elementos mínimos necesarios para el model checking[24]: un modelo, una propiedad, y una metodología de verificación.

2.6.2. La herramienta: PRISM

La herramienta de model checking que utilizaremos para el modelado y la descripción de las propiedades de sistemas es **PRISM**, un model checker simbólico que permite la verificación de sistemas probabilísticos y, en una versión reciente y todavía no estable, temporizados. Los modelos probabilísticos soportados por la herramienta son *cadenas de Markov de tiempo discreto* (DTMC), *procesos de decisión de Markov* (MDP), *cadenas de Markov de tiempo contínuo* (CTMC), y *autómatas probabilistas temporizados* (PTA). Las propiedades son especificadas usando PCTL, CSL (Continuous Stochastic Logic, usada para los CTMC), o PTCTL, de acuerdo al modelo probabilista que se esté usando.

En esta sección describiremos la sintaxis básica de PRISM necesaria para entender el lenguaje del model checker desarrollado en este trabajo, junto con algunas características de la herramienta. Dado que el modelo probabilista que usaremos son los PTA, nos enfocaremos en la descripción de los mismos y la especificación de propiedades en PTCTL.

El lenguaje de PRISM

En PRISM, un sistema está constituído por uno o más módulos que actúan de manera concurrente. Cada uno de estos módulos posee una o más *variables privadas* que representan el estado interno del módulo y un conjunto de relojes. Además, poseen *transiciones*, que especifican los posibles cambios de estado del módulo, e invariantes sobre los estados internos, que marcan restricciones en el comportamiento de acuerdo al valor de los relojes.

Comencemos con el código de ejemplo de la figura 2.5.

```
pta

module M
  s: [0..5] init 0;
  x: clock;

  invariant
    (s = 0 => x<=5)
  endinvariant

  [] s < 5 -> (s' = s+1);
  [] s = 5 -> (s' = 0) & (x'=0);
endmodule
```

Figura 2.5: Ejemplo de un PTA simple descripto con el lenguaje de PRISM

En la primera línea, declaramos el tipo de modelo probabilista que estamos usando, en este caso, PTA. Luego declaramos un módulo de nombre M, cuyo estado interno está representado por la variable `s` que es un número entero entre 0 y 5, y que en el estado inical vale 0. Además, el módulo declara la "variable" `x` que es un reloj. Lo próximo que se declara es el invariante, que establece que el módulo solo puede permanecer en el estado representado por `s=0` mientras el valor de `x` es menor o igual a 5. Los cambios de estado están dados por las transiciones que aparecen después del invariante, y donde se ve que `s` va incrementando su valor hasta llegar a 5, tras lo cual vuelve al valor 0.

Un módulo en PRISM entonces está compuesto por:

- Un nombre alfanumérico único en el sistema,
- Un conjunto de variables privadas o internas, que definen el estado interno del módulo,
- Un conjunto (posiblemente vacío) de relojes que regulan el comportamiento del módulo con restricciones temporales,

- Una condición invariante, que especifica restricciones sobre el valor de los relojes en los distintos estados, y
- Un conjunto de transiciones guardadas y posiblemente etiquetadas que definen la forma en la que el módulo cambia de estado interno.

Las variables pueden ser del tipo entero o booleano. En el caso de las variables enteras, debe especificarse el rango numérico que abarcan. Además, en ambos casos debe indicarse el valor inical de la variable.

Los relojes se declaran como cualquier otra variable, aunque deben ser de tipo `clock` y no pueden ser inicializadas, ya que por defecto su valor inicial es 0.

El invariante se declara entre las palabras claves `invariant...endinvariant`, y es una conjunción de implicaciones booleanas, en cada una de las cuales el antecedente es una condición sobre las variables de estado y el consecuente es una condición sobre los relojes.

Las transiciones constan de 3 partes:

1. Una etiqueta, que se especifica entre corchetes. La misma es usada al momento de sincronizar módulos, ya que dos módulos con transiciones con etiquetas iguales deben efectuar las mismas en el mismo instante de tiempo. La etiqueta no es obligatoria al momento de definir una transición.

2. Una guarda, que sirve para definir el conjunto de estados desde el cual es posible tomar la transición y las restricciones sobre el conjunto de relojes que especifican en que momentos está habilitada la transición.

3. Una acción probabilista, que consta de las distintas elecciones probabilistas al tomar la transición, la definición del nuevo estado interno al cual se llega por cada una de éstas, y el posible reseteo de un subconjunto de los relojes del módulo.

Por ejemplo, si consideramos la siguiente transición, donde `buff` y `error` son variables privadas y `x` es un reloj:

```
[send] buffer > 0 & x >= 2 -> 0.99: (buffer'=buffer-1) & (x'=0)
                            + 0.01: (error'=true);
```

vemos que la etiqueta de la misma es `send`. La acción se puede tomar en cualquier estado donde `buff>0`, siempre y cuando el valor del reloj `x` sea mayor o igual a 2. En caso de que la transición se tome, hay 2 resultados posibles: con probabilidad 0,99 el valor de `buff` se decrementa y `x` se resetea; y con probabilidad 0,01, el valor de `error` cambia a `true`.

En el caso más complejo de la figura 2.6 tenemos dos módulos interactuando, que representan el comportamiento de una impresora y el driver de

```
pta

module PrinterDriver
  buff: [0..10] init 0;
  x: clock;

  [] buff<10 -> (buff'=buff+1);
  [sendToPrinter] buff>0 & x>=1 -> (buff'=buff-1) & (x'=0);
endmodule

module Printer
  queue: [0..10] init 0;
  inklow: bool init false;
  y: clock;

  invariant
    (queue>0 => y<=5)
  endinvariant

  [sendToPrinter] queue<10 -> 0.95: (queue'=queue+1) & (y'=0)
                            + 0.05: true;
  [] queue>0 & y >=2 -> (queue'=queue-1) & (y'=0);
  [] !inklow -> (inklow'=true);
endmodule
```

Figura 2.6: Un modelo PRISM más complejo, con 2 modulos que sincronizan en la acción **sendToPrinter**.

la misma. El módulo `PrinterDriver` envía documentos a la cola de impresión de `Printer`. La restricción impuesta por el reloj `x` implica que entre dos envíos consecutivos debe transcurrir al menos una unidad de tiempo. Notar que cuando `0<buff<10`, el módulo `PrinterDriver` tiene dos transiciones para tomar (i.e. una elección no-determinista). El módulo `Printer` se comporta de manera parecida, y también incorpora comportamiento temporal con el reloj `y`. Ambos módulos poseen una transición con la etiqueta `sendToPrinter`, lo que indica que deben efectuar la misma al mismo tiempo. Además, cuando `Printer` realiza la transición `sendToPrinter`, hay dos resultados posibles: con probabilidad 0,95 el documento se agrega a la cola de impresión, mientras que con probabilidad 0,05, nada ocurre, es decir, el documento no se agrega a la cola (aunque sí desaparece del buffer de `PrinterDriver`). Finalmente, `Printer` tiene también un indicador booleano del nivel de tinta de la impresora, `inklow`, que en cualquier momento puede cambiar su valor a `true`, indicando que el nivel de tinta es bajo (aunque, en este modelo, el

nivel de tinta no affecta el comportamiento de la impresora).

PRISM también permite la declaración de *constantes* y *fórmulas*. Las constantes se declaran fuera de los módulos, anteponiendo la palabra clave `const`, y pueden ser de tipo `int`, `double`, o `bool`. Por ejemplo, para declarar una constante *pi* de tipo `double` que valga 3,14159, escribimos

```
const double pi = 3.14159;
```

Las fórmulas son formas de "nombrar" una expresión booleana para que no sea necesario reescribirla cada vez que la usamos. Además, como veremos después, sirven también para darle más sentido y legibilidad a las propiedades. Las fórmulas se declaran fuera de los módulos, anteponiendo la palabra clave `formula`:

```
formula queue_almost_full = (queue_size >= 8);
```

Las *propiedades* de los PTA se declaran usando la lógica temporal PTCTL. Las mismas deben estar en un archivo distinto al de los módulos. Es decir, al describir un sistema en PRISM, usaremos generalmente dos archivos: uno para el modelo, y otro para las fórmulas. Dado que la versión de PRISM usada para el desarrollo de este trabajo es una versión todavía en desarrollo, nos encontramos con varias limitaciones al momento de especificar propiedades. Las mismas solo pueden ser de la forma

```
Pmax=? [ formula_de_camino ]
```

```
Pmin=? [ formula_de_camino ]
```

Donde `formula_de_camino` a su vez debe ser de la forma:

```
F<t formula_de_estado
```

```
F<=t formula_de_estado
```

con `t` un número entero y `formula_de_estado` tal que **no** puede contener operadores `Pmax` o `Pmin`, es decir, no se permiten operadores `Pmax` o `Pmin` anidados. El operador `Pmax=?` calcula la máxima probabilidad (entre todos los adversarios) de que un camino que comience en el estado inicial cumpla una propiedad; el operador `Pmin=?` hace lo mismo pero calculando la probabilidad mínima. Volviendo al ejemplo de la impresora, la propiedad

```
Pmax=? [ F<=30 queue=10 ]
```

calcula la probabilidad máxima de que, en a lo sumo 30 unidades de tiempo, la cola de impresión se llene.

Uso de la herramienta

Para ejecutar la verificación necesitaremos primero de dos archivos: uno donde esté guardado el modelo del sistema, y otro donde estén guardadas las propiedades, digamos `modelo.nm` y `prop.pctl`, respectivamente.

Las extensiones son solo simbólicas. Las mismas tienen origen en las extensiones usadas para otros modelos probabilísticos de PRISM: `.nm` para MDP y `.pctl` para propiedades descriptas en PCTL.

Necesitaremos además, obviamente, el model checker PRISM. El mismo puede descargarse en `http://www.prismmodelchecker.org` de forma gratuita y se distribuye bajo la licencia GPL[1]. Además, al estar implementado en Java[2], la herramienta es multiplataforma.

Supongamos que el directorio donde se encuentra el código fuente de PRISM se llama `prism` y está en nuestra carpeta personal. Para iniciar la verificación del modelo antes mencionado, debemos ejecutar el siguiente comando:

```
nico@halibel:~$ ./prism/bin/prism modelo.nm prop.pctl
```

El model checker además tiene bastantes opciones, las cuales se pueden ver con la opción `-help`. Sin embargo, al verificar propiedades sobre PTA, la opción más importante es sin duda `-ptamethod`, la cual permite elegir la metodología usada para realizar la verificación. Las opciones de las que disponemos son `games` y `digital`. La primera hace referencia al motor *Abstraction Refinement*, cuya información puede encontrarse en [20], y que es la opción por defecto. La segunda se refiere al motor *Digital Clocks*, descripto en [21]. Si bien los detalles de ambos están fuera del alcance de este trabajo, podemos resumir que Abstraction Refinement trabaja con un modelo *denso* del tiempo, mientras que Digital Clocks los hace con un modelo *discreto*. Éste último introduce algunas limitaciones en la descripción de los modelos y propiedades, por ejemplo, no es posible usar relaciones estrictas (las que usan $<$ o $>$) en las restricciones de reloj. Sin embargo, para los sistemas que pueden modelarse incluso con esta restricción, Digital Clocks es probablemente la mejor opción. A pesar de las numerosas mejoras en la eficiencia del motor Abstraction Refinement[31, 22], la complejidad agregada por los relojes suele desbordar facilmente la capacidad de cualquier computadora ordinaria.

[2] http://www.java.com/

Capítulo 3

Sistemas Tolerantes a Fallas

El concepto de *toleracia a fallas* existe desde hace mucho tiempo. Se refiere a la capacidad de una pieza de hardware o software para continuar prestando las funcionalidades requeridas a pesar de la ocurrencia de fallas ocacionales, ya sean pasajeras o permanentes, en módulos y componentes internos[27].

La creciente complejidad tanto del hardware como del software genera que los mismos sean cada vez más propensos a fallas. Sin embargo, el hardware suele presentar una complejidad notablemente menor. La razón es que, en general, las piezas de hardware poseen una cantidad relativamente pequeña de estados internos, mientras que algunos simples sistemas de software llegan a tener una inmensa cantidad de los mismos. Además, la tolerancia a fallas sobre el hardware se aplica generalmente a rupturas o al mal funcionamiento de componentes, mientras que en el software, se aplica exclusivamente a errores de diseño, que es de donde provienen las fallas. Y si bien hoy en día existen numerosas técnicas para detectar y reducir el número de fallas en los sistemas que se desarrollan, esto no es suficiente: se necesita que los sistemas acepten la ocurrencia de ciertas fallas e incorporen técnicas para tolerar las mismas y brindar un nivel de funcionalidad aceptable[19].

La tolerancia a fallas es vital en sistemas críticos que operan en entornos:

- donde las fallas son frecuentes y/o inevitables,
- donde el acceso es dificultoso, costoso o peligroso,
- donde un error producido por falla no es aceptable.

Es así que, por ejemplo, en sistemas de control y medición en aviones, sistemas espaciales, equipo médico, o maquinaria operando en zonas de radioactividad, no se puede permitir que una falla produzca un resultado no esperado, ya que esto puede derivar en graves consecuencias. La tolerancia a fallas es necesaria para reducir tanto los riesgos potenciales como los riesgos ocultos de los sistemas tecnológicos, a un nivel que pueda ser considerado seguro[18].

En este capítulo hablaremos del comportamiento de los sistemas tolerantes a fallas, viéndolos como máquinas de estados y transiciones, donde los estados pueden ser "riesgosos" o "seguros" (de acuerdo a si el sistema está operando bajo el efecto de una falla o no), y describiremos la idea del proceso de model checking utilizada en la realización de este trabajo.

3.1. Comportamineto de un Sistema Tolerante a Fallas

Al momento de modelar el comportamiento de un sistema, el mismo puede ser visto como una máquina de estados: un conjunto de estados internos y transiciones entre estos estados. Los estados representan información sobre el sistema y sus componentes, mientras que las transiciones hacen lo propio con las acciones efectuadas por el sistema.

Las fallas, sin embargo, no deben considerarse como *parte* del sistema, ya que representan un comportamiento anómalo o no planeado del mismo. A pesar de esto, si se quieren verificar propiedades sobre el modelo del sistema tomando en cuenta la posible ocurrencia de una o más fallas, es necesario que la mismas sean, de alguna forma, parte del modelo. Esta metodología de incorporar las fallas al modelo para determinar la respuesta del sistema a las mismas se denomina *fault-injection*[4, 29, 10].

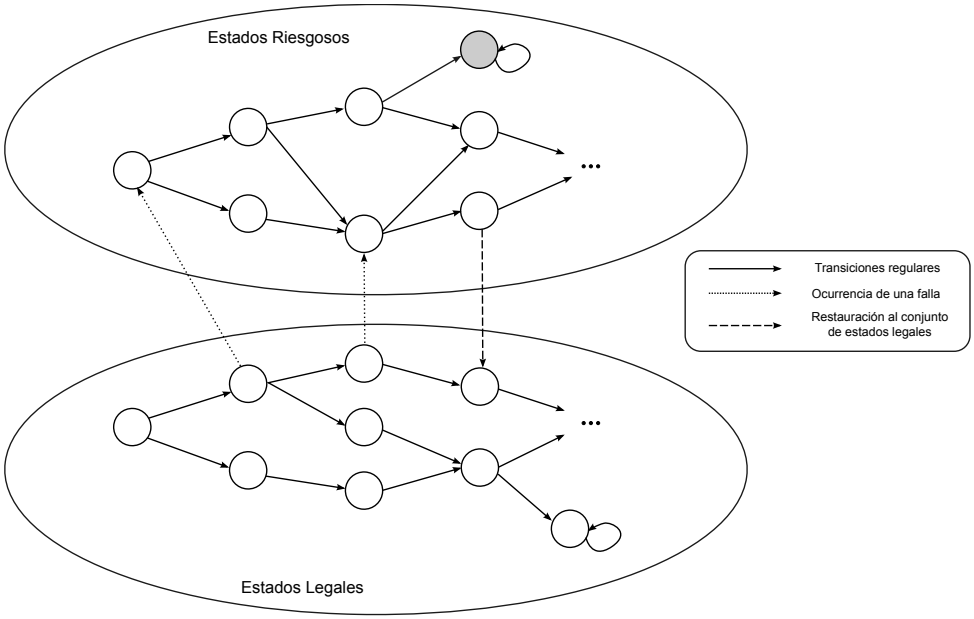


Figura 3.1: Estados y transiciones de un sistema tolerante a fallas

En nuestro caso, representaremos la ocurrencia de una falla como una transición anómala en el modelo del sistema. Distinguiremos entonces dos

conjuntos de estados: los estados "legales" y los estados "riesgosos"[16]. La ocurrencia de una falla puede provocar que el sistema abandone el conjunto de estados legales y comience a operar sobre el conjunto de estados riesgosos. En la figura 3.1 podemos ver un esquema de este comportamiento. Cuando ocurre una falla, el sistema comienza a ejecutar las transiciones del conjunto de estados riesgosos. En algunos casos, no será necesario volver al conjunto de estados legales, pues el sistema podrá seguir operando de manera correcta incluso en presencia de la falla. En otros casos, el sistema ejecutará una cantidad finita de transiciones en el entorno con fallas antes de volver al conjunto de estados legales para así continuar operando. En cualquiera de los dos casos anteriores, se dice que el sistema tolera la falla. Existe, sin embargo, la posibilidad de que el sistema llegue a un estado riesgoso desde el cual no exista forma de volver al conjunto de estados legales ni de continuar su ejecución normal (en la figura, es el caso del estado de color gris). Si esto ocurre, entonces diremos que el sistema no tolera la falla.

3.2. Model Checking de Sistemas Tolerantes a Fallas

El proceso de model checking para sistemas tolerantes a fallas que proponemos varía un poco del proceso descripto en la sección 2.6.1, pues, como dijimos antes, la ocurrencia de una falla no debería ser vista como una transición del sistema. Lo que proponemos entonces es modelar las fallas y los módulos del sistema de manera independiente. En un paso intermedio, la herramienta incorporará las fallas al modelo del sistema antes de verificar las propiedades especificadas.

En la figura 3.2 podemos ver un esquema similar al de la sección 2.6.1, pero en este caso con la incorporación de las fallas. El proceso, dado un sistema, un conjunto de requerimientos y un conjunto de fallas, consiste en los siguientes pasos:

1. Dar un modelo del sistema en el lenguaje de la herramienta

2. Especificar los requerimientos (teniendo en cuenta la ocurrencia de fallas) como propiedades en una lógica que la herramienta entienda.

3. Especificar las fallas en el lenguaje de la herramienta

4. Obtener, mediante un paso intermedio de la herramienta, un modelo donde estén presentes tanto las transiciones del sistema como las que representan la ocurrencia de fallas.

5. Verificar la propiedades especificadas contra el modelo obtenido en el paso anterior.

Cabe mencionar que las propiedades especificadas para sistemas tolerantes a fallas son, por lo general, diferentes de las especificadas para sistemas que no lo son. En el primer caso, la posibilidad de ocurrencia de fallas es relevante al momento de establecer los requerimentos, por lo tanto, esto se ve reflejado en las propiedades.

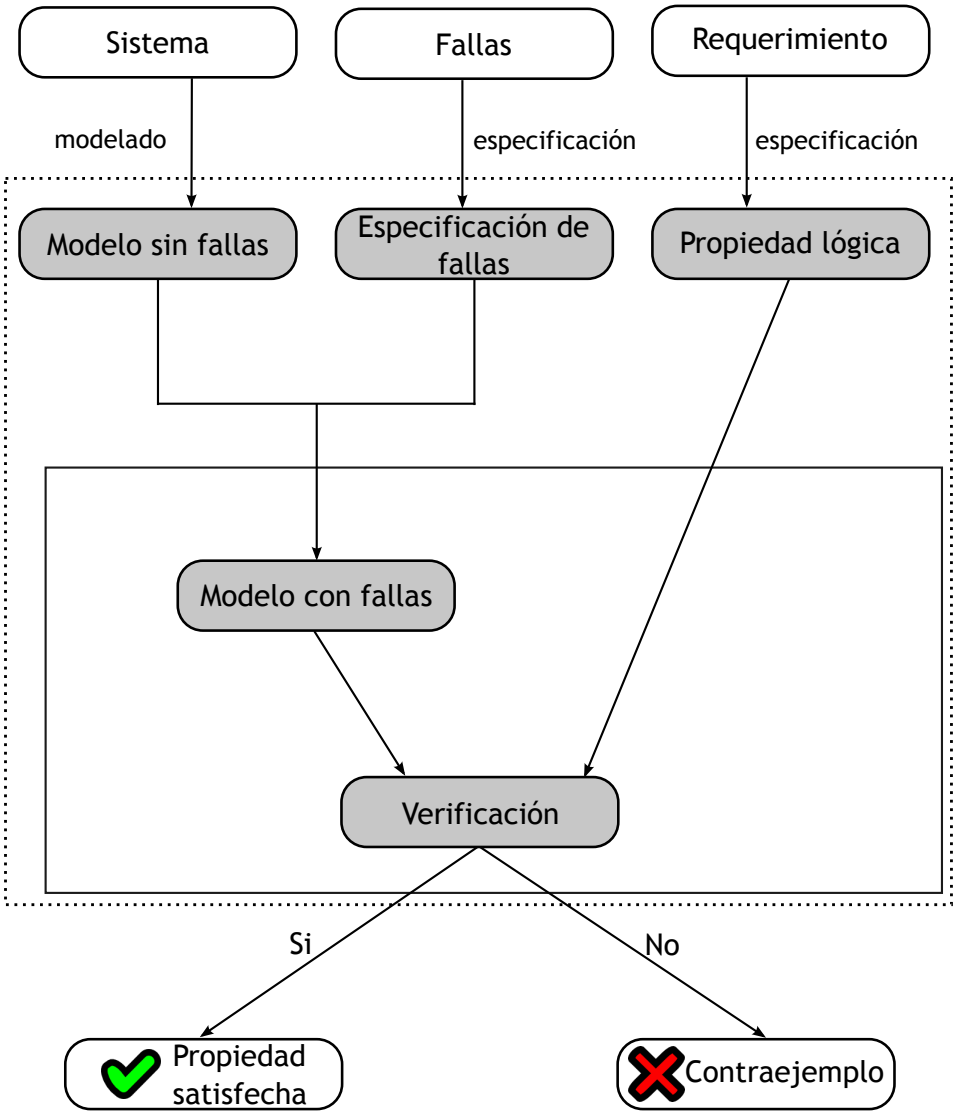


Figura 3.2: Nuestro proceso de model checking en sistemas tolerantes a fallas

Capítulo 4

Modelando Sistemas Tolerantes a Fallas

A lo largo de este capítulo veremos el modelo usado para describir el comportamiento de los sistemas tolerantes a fallas. Dado un PTA representando un módulo del sistema y un conjunto de fallas a las cuales ese módulo es sensible, seremos capaces de generar un conjunto de autómatas probabilistas temporizados que, en ejecución concurrente, modelarán el comportamiento del módulo considerando la ocurrencia de fallas.

4.1. Fallas

En la sección 3.1 vimos que el comportamiento de un sistema tolerante a fallas difiere del de un sistema regular al incluir un conjunto de estados a los cuales se accede cuando uno o más módulos se encuentran bajo el efecto de fallas. En nuestro modelo, las fallas no definiran explícitamente estos estados, sino que los mismos serán deducibles en base a la información de la misma. Dado un autómata probabilista temporizado que representa el comportamiento de un módulo de un sistema, una falla será un efecto sobre el modelo, que deshabilitará algunas transiciones, habilitará algunas nuevas, y producirá cambios en el nodo actual y en los relojes del PTA en el momento que ocurra. La ocurrencia de la falla se dará con restricciones probabilísticas y temporales, es decir, ocurrirá con cierta probabilidad, dentro de un intervalo de tiempo. Además, la falla tendrá asociada una condición de restauración, que servirá para determinar cuándo el módulo se ha recuperado de la misma.

Dado un autómata probabilista temporizado

$$PTA = \langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob \rangle$$

podemos entonces definir formalmente una falla como una 6-upla

$$\mathcal{F}_{PTA} = \langle Pre, Prob, Effect, Disables, Restores, Step \rangle$$

tal que:

- $Pre \subseteq L$ es la *precondición* de la falla,
- $Prob : \mathbb{R}^+ \to [0, 1]$, es la *probabilidad* de ocurrencia de la falla,
- $Effect \subseteq Pre \times Dist(2^{\mathcal{X}} \times L)$ es el *efecto* de la falla, tal que

$$Pre \subseteq \bigcup_{(Pre', p') \in Effect} Pre'$$

- $Disables \subseteq prob$ es el conjunto de *transiciones deshabilitadas* por la falla,
- $Restores \subseteq L$ es la *condición de restauración* de la falla,
- $Step \in \mathbb{N}$ es el *intervalo de tiempo* de la ocurrencia de la falla.

Intuitivamente, la falla solo puede ocurrir mientras el PTA se mueva en el conjunto de estados *Pre*. Si es así, la falla $\mathcal{F}_{PTA}$ puede "intentar" ocurrir una vez por cada *Steps* unidades de tiempo. En ese intento, la probabilidad de que la falla efectivamente ocurra es $Prob(t)$, donde t es el tiempo transcurrido desde la última vez que el PTA se movió desde un nodo $l \notin Pre$ a un nodo $l' \in Pre$. Si la falla ocurre estando en un nodo l, entonces el PTA debe tomar en ese momento una de las transiciones de $Effect(l)$. Además, a partir de ese momento, diremos que $\mathcal{F}_{PTA}$ está *activa*. Mientras la falla esté activa, el sistema solo podrá ejecutar las transiciones del conjunto $prob - Disables$. Si, luego de una cantidad finita de transiciones el sistema llega a un estado $l' \in Restores$, diremos entonces que la falla ya no está activa, y el sistema podrá ejecutar nuevamente todas las transiciones de *prob*.

Si bien *Effect*, según como lo definimos, es muy flexible al momento de definir las transiciones que puede tomar cada nodo en el caso de la ocurrencia de la falla, esto no será así en la implementación de la herramienta, ya que el efecto deberá definirse como una función del nodo en el cual ocurre la falla, donde dicho nodo está dado por el valor de las variables del módulo. Esto implica que las transiciones del efecto de la falla correspondientes a distintos nodos serán muy similares entre si. Esto se explicará con más detalle en la sección 5.1. Además, si bien según la definición de *Effect* a cada nodo de *Pre* le corresponden una o más distribuciones sobre $2^{\mathcal{X}} \times L$, donde L es el conjunto de nodos del PTA, hay que notar que esta definición se hizo en base al modelado correspondiente a la herramienta de model checking y no en base al modelo teórico. Es decir, hay que tener en cuenta que la inyección de fallas puede agregar estados al modelo teórico (estados "riesgosos", no considerados en el modelo sin fallas). Por ejemplo, si tenemos un PTA que representa el comportamiento de dos variables x e y en un programa en el que siempre se cumple el invariante $x \leq y$, los nodos en los cuales $x > y$ no

estarán en L. Sin embargo, la ocurrencia de una falla puede llevar al PTA a un nodo donde el invariante no sea válido. Distinto es el caso del modelo correspondiente a la herramienta de model checking, donde todos los nodos existen (incluso aquellos donde no se cumple el invariante) solo que algunos no son alcanzables.

Para lidiar con algunas restricciones impuestas en la definición de los autómatas probabilistas temporizados, solo permitiremos que *Prob* sea una función *escalonada*, donde los cambios de valor se den en instantes enteros de tiempo. Definiremos además la secuencia

$$Seq(Prob) = (time_0, prob_0), \ldots, (time_n, prob_n)$$

tal que

- $time_i \in \mathbb{N}_0$ para todo $i \in [0, n]$,
- $prob_i \in [0, 1]$ para todo $i \in [0, n]$,
- $time_0 = 0$,
- $time_i < time_{i+1}$ para todo $i \in [0, n-1]$,
- Para $i < n$, $Prob(t) = prob_i$ para todo $t \in [time_i, time_{i+1})$,
- Para $i = n$, $Prob(t) = prob_i$ para todo $t \in [time_i, \infty)$,

Es decir, $Seq(Prob)$ es otra forma de definir *Prob*, que nos será útil al momento de describir al autómata probabilista temporizado asociado con la falla, que veremos en la siguiente sección.

4.2. Módulos

En este trabajo modelaremos un módulo como un autómata probabilista temporizado que tiene asociado un conjunto finito de fallas. Es decir, nuestro modelo será una estructura:

$$\mathcal{M} = \langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob, F \rangle$$

donde $PTA_{\mathcal{M}} = \langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob \rangle$ es un PTA y $F = \{\mathcal{F}_0, \ldots, \mathcal{F}_n\}$ es un conjunto de fallas asociadas al PTA. La semántica de $\mathcal{M}$ será un autómata probabilista temporizado

$$PTA_{\mathcal{M},F} = PTA'_{\mathcal{M}} \parallel PTA_{\mathcal{M},\mathcal{F}_0} \parallel \ldots \parallel PTA_{\mathcal{M},\mathcal{F}_n}$$

donde $PTA'_{\mathcal{M}}$ es una variación del PTA asociado al modelo del módulo, y para cada $i \in [0, n]$, $PTA_{\mathcal{M},\mathcal{F}_i}$ es un PTA representando el comportamiento de cada una de las fallas.

4.2.1. Módulo Principal

El PTA del módulo principal, $PTA'_{\mathcal{M}}$, se crea a partir de $PTA_{\mathcal{M}}$, agregando comportamiento apropiado en la ocurrencia de una o más fallas. Los nodos de $PTA'_{\mathcal{M}}$ son similares a los de $PTA_{\mathcal{M}}$, salvo que en el primero cada uno de estos nodos esta asociado al "estado" de las fallas. Además, a algunos de estos nodos los denominaremos *nodos de salto*, y serán nodos deben ser abandonados en el mismo instante en el que se entra a ellos. Estos nodos servirán como nodos de transición, cuando alguno de los demás nodos cambie de estado. Formalmente, sean

$$PTA_{\mathcal{M}} = \langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob \rangle$$

$$F = \{\mathcal{F}_0, \ldots, \mathcal{F}_n\}$$

tenemos que

$$PTA'_{\mathcal{M}} = \langle L', \mathcal{L}', \bar{l}', Act', \mathcal{X}', inv', prob' \rangle$$

El conjunto de nodos L' está compuesto por los nodos de L asociados al "estado" de las fallas. Sea $st : F \to \{0, 1, 2, 3\}$ una función que indica en que estado se encuentra cada falla. Llamaremos a este tipo de funciones *valuaciones* de F. Los estados representan, intuitivamente, lo siguiente:

- 0: La falla no puede ocurrir, pues no se cumple la precondición.
- 1: La falla puede ocurrir, pero todavía no ocurrió (es decir, se cumple la precondición).
- 2: La falla está haciendo un intento de ocurrir, aunque esto no implica que efectivamente vaya a ocurrir. Los nodos con alguna falla en este estado son algunos de los nodos de salto.
- 3: La falla está activa.

Además, definiremos la función

$$st[\mathcal{F} := n](\mathcal{G}) = \begin{cases} n & \mathcal{G} = \mathcal{F} \\ st(\mathcal{G}) & \mathcal{G} \neq \mathcal{F} \end{cases}$$

Luego, el conjunto de estados L' es tal que

$$L' = L \times \{\ st \mid st \text{ es una valuación de } F\ \}$$

La función $\mathcal{L}'$ será tal que $\mathcal{L}'(l, st) = \mathcal{L}(l)$. Además $\bar{l}' = (\bar{l}, zero)$ donde $zero(\mathcal{F}) = 0$ para todo $\mathcal{F} \in F$. En el conjunto Act' agregaremos las etiquetas propias de cada falla, necesarias para la composición paralela:

$$
\begin{aligned}
Act' = Act \cup \{ \quad & wake_{\mathcal{M},\mathcal{F}}, \\
& sleep_{\mathcal{M},\mathcal{F}}, \\
& try_{\mathcal{M},\mathcal{F}}, \\
& fail_{\mathcal{M},\mathcal{F}}, \\
& dontfail_{\mathcal{M},\mathcal{F}}, \\
& restorewake_{\mathcal{M},\mathcal{F}}, \\
& restoresleep_{\mathcal{M},\mathcal{F}} \mid \mathcal{F} \in F \ \}
\end{aligned}
$$

El nuevo conjunto de relojes $\mathcal{X}'$ agregará solo un reloj, necesario para poder salir de los nodos de salto en el mismo momento en el que se ingresa a éstos. Por lo tanto, $\mathcal{X}' = \mathcal{X} \cup \{ \ z_{\mathcal{M}} \ \}$, donde $z_{\mathcal{M}}$ es el nuevo reloj. Así mismo, la función de invariante inv' agregará solo las condiciones necesarias para la salida inmediata en los nodos de salto. Formalmente,

$$
inv'(l, st) = \begin{cases} inv(l) & \bigwedge_{\mathcal{F} \in F} \neg jump_{\mathcal{F}}(l, st) \\ z_{\mathcal{M}} \leq 0 & \bigvee_{\mathcal{F} \in F} jump_{\mathcal{F}}(l, st) \end{cases}
$$

donde

$$
\begin{aligned}
jump_{\mathcal{F}}(l, st) = \quad & (l \in Pre_{\mathcal{F}} \wedge st(\mathcal{F}) = 0) \\
& \vee (l \notin Pre_{\mathcal{F}} \wedge st(\mathcal{F}) = 1) \\
& \vee st(\mathcal{F}) = 2 \\
& \vee (st(\mathcal{F}) = 3 \wedge l \in Restores_{\mathcal{F}})
\end{aligned}
$$

$jump_{\mathcal{F}}$ representa la condición de salto asociada a la falla $\mathcal{F}$. Notar que, al estar definida como una disjunción, podemos distinguir 4 casos:

1. Cuando el PTA está en un estado que "cumple" la precondición de la falla, pero según la función st la falla todavía no puede ocurrir (es decir, le asigna a $\mathcal{F}$ el valor 0). En este caso, debemos "saltar" para cambiar inmediatamente el valor de $st(\mathcal{F})$.

2. Al contrario del caso anterior, también debemos saltar si el PTA llega a un estado que no cumple la precondición de la falla, pero donde st indica que la falla puede ocurrir (es decir, $st(\mathcal{F}) = 1$).

3. Si llegamos a un estado donde $st(\mathcal{F}) = 2$, sabemos que estamos en un nodo de transición (decidiendo si la falla ocurre o no).

4. Si la falla está activa, pero el PTA llega a un estado que cumple la condición de restauración, debemos cambiar inmediatamente a un estado en el cual la falla no esté activa.

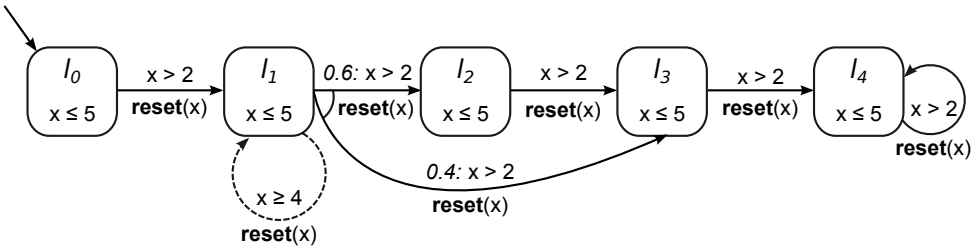
Por último, resta definir el conjunto $prob'$. El mismo agregará las transiciones en los nodos de salto, y también restringirá las transiciones de $prob$ en el caso de que alguna falla esté activa. También especificará las guardas y reseteos sobre el nuevo reloj $z_{\mathcal{M}}$.

Formalmente,

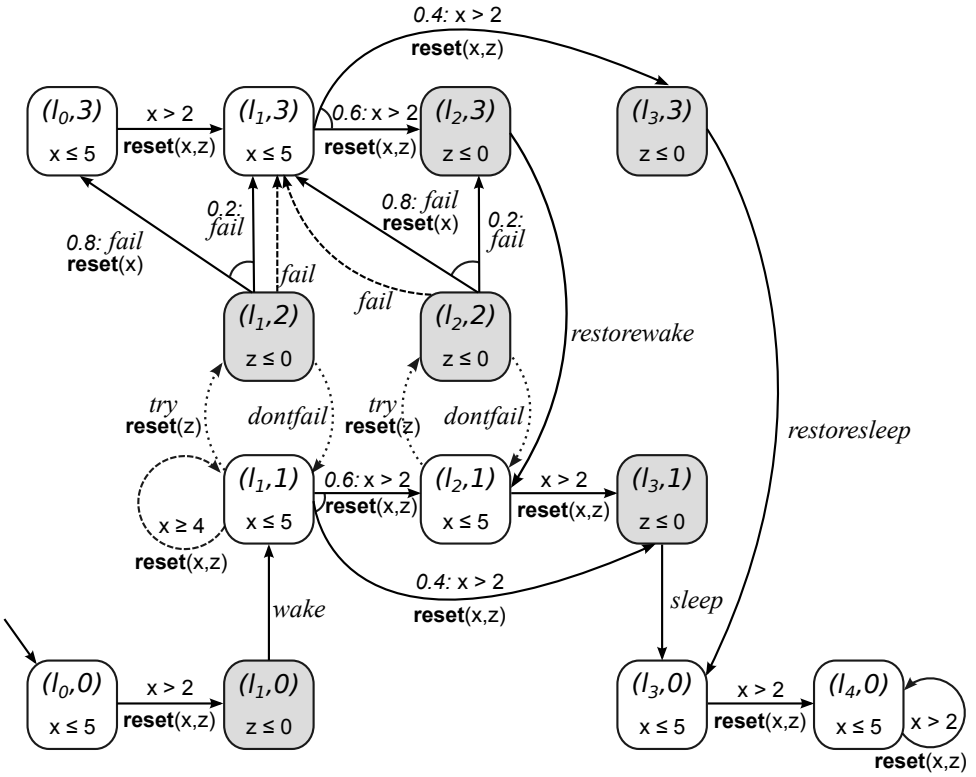
$$
\begin{aligned}
prob' \;=\; & \{ \;\; ((l,st), true, wake_{\mathcal{M},\mathcal{F}}, p) \text{ tal que} \\
& \quad \mathcal{F} \in F \wedge \; st(\mathcal{F}) = 0 \wedge \; l \in Pre_{\mathcal{F}} \wedge \; p(\emptyset, (l, st[\mathcal{F} := 1])) = 1 \;\} \qquad (1) \\
\cup \; & \{ \;\; ((l,st), true, sleep_{\mathcal{M},\mathcal{F}}, p) \text{ tal que} \\
& \quad \mathcal{F} \in F \wedge \; st(\mathcal{F}) = 1 \wedge \; l \notin Pre_{\mathcal{F}} \wedge \; p(\emptyset, (l, st[\mathcal{F} := 0])) = 1 \;\} \qquad (2) \\
\cup \; & \{ \;\; ((l,st), true, try_{\mathcal{M},\mathcal{F}}, p) \text{ tal que} \\
& \quad \mathcal{F} \in F \wedge \; st(\mathcal{F}) = 1 \wedge \; l \in Pre_{\mathcal{F}} \wedge \\
& \quad p(\{z_{\mathcal{M}} = 0\}, (l, st[\mathcal{F} := 2])) = 1 \;\} \qquad (3) \\
\cup \; & \{ \;\; ((l,st), true, fail_{\mathcal{M},\mathcal{F}}, p) \text{ tal que} \\
& \quad \mathcal{F} \in F \wedge \; st(\mathcal{F}) = 2 \wedge l \in Pre_{\mathcal{F}} \wedge \\
& \quad \exists p' \in Effect_{\mathcal{F}}(l) : \forall X, l' : p(X, (l', st[\mathcal{F} := 3])) = p'(X, l') \;\} \qquad (4) \\
\cup \; & \{ \;\; ((l,st), true, dontfail_{\mathcal{M},\mathcal{F}}, p) \text{ tal que} \\
& \quad \mathcal{F} \in F \wedge \; st(\mathcal{F}) = 2 \wedge \; p(\emptyset, (l, st[\mathcal{F} := 1])) = 1, \forall \mathcal{F} \in F \;\} \qquad (5) \\
\cup \; & \{ \;\; ((l,st), true, resotorewake_{\mathcal{M},\mathcal{F}}, p) \text{ tal que} \\
& \quad \mathcal{F} \in F \wedge \; st(\mathcal{F}) = 3 \wedge \; l \in Restores_{\mathcal{F}} \wedge l \in Pre_{\mathcal{F}} \\
& \quad \wedge \, p(\emptyset, (l, st[\mathcal{F} := 1])) = 1 \;\} \qquad (6) \\
\cup \; & \{ \;\; ((l,st), true, resotoresleep_{\mathcal{M},\mathcal{F}}, p) \text{ tal que} \\
& \quad \mathcal{F} \in F \wedge \; st(\mathcal{F}) = 3 \wedge \; l \in Restores_{\mathcal{F}} \wedge l \notin Pre_{\mathcal{F}} \\
& \quad \wedge \, p(\emptyset, (l, st[\mathcal{F} := 0])) = 1 \;\} \qquad (7) \\
\cup \; & \{ \;\; ((l,st), g, \alpha, p') \text{ tal que} \\
& \quad (l, g, \alpha, p') \in prob \setminus \textstyle\bigcup_{st(\mathcal{F})=3} Disables_{\mathcal{F}} \wedge \\
& \quad \textstyle\bigwedge_{\mathcal{F} \in F} (\neg jump_{\mathcal{F}}(l, st) \wedge st(\mathcal{F}) \neq 2) \wedge \\
& \quad \forall X, l' : p'(X \cup \{z_{\mathcal{M}}\}, (l', st)) = p(X, l') \;\} \qquad (8)
\end{aligned}
$$

En la figura 4.1a vemos el ejemplo de un PTA simple, con 5 nodos. Notar que el nodo l_1 tiene dos transiciones salientes, una de las cuales tiene además una elección probabilística. En la figura 4.1b podemos ver el autómata probabilista temporizado anterior, luego de agregar todos los cambios para adaptarlo a la ocurrencia de una falla $\mathcal{F}$, tal que:

- $Pre_{\mathcal{F}} = \{l_1, l_2\}$
- $Restores_{\mathcal{F}} = \{l_2, l_3\}$
- $Disables_{\mathcal{F}} = \{(l_1, x \geq 4, \tau, p)\}$, con $p(\{x\}, l_1) = 1$
- $Effect_{\mathcal{F}} = \{(l_1, p_1), (l_1, p_2), (l_2, q_1), (l_2, q_2)\}$, con:
  - $p_1(\{x\}, l_0) = q_1(\{x\}, l_1) = 0{,}8$



(a)



(b)

Figura 4.1: Un PTA, antes y después de la aplicación del modelo de fallas

- $p_1(\emptyset, l_1) = q_1(\emptyset, l_2) = 0{,}2$
- $p_2(\emptyset, l_2) = q_2(\emptyset, l_2) = 1$

Los nodos oscuros son los que denominamos nodos de salto. En el autómata probabilista temporizado de la figura 4.1b se pueden ver 4 "niveles", representados por las filas de nodos del PTA. Estos niveles están dados por el estado de la falla $\mathcal{F}$. Notar que, para que el dibujo sea más entendible, a cada nodo se le asoció un número en vez de una valuación del conjunto de fallas. Es

decir, el nodo (l, st) está representado en el dibujo por $(l, st(\mathcal{F}))$. En el nivel inferior (donde $st(\mathcal{F}) = 0$) están los nodos donde la falla no puede ocurrir (pues no se cumple la precondición), más el nodo de salto para pasar al nivel que está más arriba. Cuando $st(\mathcal{F}) = 1$, tenemos los nodos donde se cumple $Pre_{\mathcal{F}}$ pero donde la falla todavía no ha ocurrido. Notar que a los nodos de este nivel (los que no son nodos de salto) se les ha agregado la transición con la etiqueta try, que indican el intento de la falla por ocurrir. Al sincronizar este PTA con el que veremos en la próxima sección, este intento se dará solo una vez por cada $Steps_{\mathcal{F}}$ unidades de tiempo. En el nivel donde $st(\mathcal{F}) = 2$ los estados son solo estados de salto, donde se decidirá sin demora si la falla efectivamente ocurre o no. En caso de que no, se vuelve al nivel anterior. Si la falla ocurre, entonces pasamos al nivel donde la falla está activa (con $st(\mathcal{F}) = 3$) mediante una transición etiquetada con $fail$. En este modo, solo se podrán tomar las transiciones que no estén deshabilitadas por la falla (en este ejemplo, el auto-ciclo en l_1, marcado con la flecha con guiones, está deshabilitado cuando $\mathcal{F}$ está activa).

4.2.2. Módulo de Fallas

La estructura del módulo que representa una falla es genérica, es decir, todas las fallas se representan con módulos similares. Al sincronizar estos módulos con la variación del módulo principal, se obtiene el comportamiento que contempla la posible ocurrencia de fallas.están

El módulo de una falla cuenta con 6 nodos y con 1 o 2 relojes, dependiendo de la función $Prob$. Inicialmente, la falla se encuentra en el nodo l_0. En algún momento el módulo ejecutará la acción $wake$ para pasar al nodo l_1, resetando todos los relojes (luego de la sincronización con el módulo principal, esto se dará cuando el sistema pase a un estado donde la ocurrencia de la falla es posible). En este estado, tenemos la posibilidad de volver a l_0, mediante la acción $sleep$. También tenemos un conjunto de transiciones probabilísticas etiquetadas con try, que indicarán la probabilidad de ocurrencia de la falla en un determinado intervalo de tiempo. En caso de que la función $Prob$ solo defina la probabilidad para el valor de tiempo $t = 0$ (esto es, cuando $Seq(Prob) = (0, p)$), el módulo solo tendrá un reloj (el reloj x en la figura 4.2), ya que no es necesario contabilizar el tiempo transcurrido desde la habilitación de la falla; en caso contrario, habrá además un reloj y, que se iniciará cada vez que se efectúe la acción $wake$. Las transiciones etiquetadas con try pueden llevar, con distinta probabilidad, a los nodos l_2 o l_3. El primer caso indica que la falla ocurre, por lo que se debe tomar la transición $fail$ hasta el nodo l_5, y esperar allí hasta que el sistema se restaure de la falla, caso en el cual se tomará alguna de las transiciones $restorewake$ o $restoresleep$, de acuerdo a si el sistema se encuentra en un estado donde la falla puede ocurrir o no. En el caso en que la transición try lleve al nodo l_3 el módulo tomará la transición $dontfail$ hasta el nodo l_4 para indicar que la falla no

ocurre, y esperará allí hasta que la falla se inhabilite (tomando la transición *sleep*) o hasta que finalice el intervalo de *Steps* unidades de tiempo, luego del cual regresará al nodo l_1.

Formalmente, dada una falla $\mathcal{F}$ tal que

$$Seq(Prob_{\mathcal{F}}) = (0, p_0), (t_1, p_1), \ldots, (t_n, p_n)$$

es decir, tiene al menos un cambio de probabilidad, el $PTA_{\mathcal{M},\mathcal{F}}$ es una estructura $\langle L, \mathcal{L}, \bar{l}, Act, \mathcal{X}, inv, prob \rangle$ tal que

- $L = \{l_0, l_1, \ldots, l_5\}$
- $\bar{l} = l_0$
- $Act = \left\{ \begin{array}{l} wake_{\mathcal{M},\mathcal{F}}, sleep_{\mathcal{M},\mathcal{F}}, try_{\mathcal{M},\mathcal{F}}, fail_{\mathcal{M},\mathcal{F}}, dontfail_{\mathcal{M},\mathcal{F}}, \\ restoresleep_{\mathcal{MF}}, restorewake_{\mathcal{M},\mathcal{F}} \end{array} \right\}$
- $\mathcal{X} = \{x, y\}$
- $inv(l) = \left\{ \begin{array}{ll} x < Steps & l = l_1 \\ x \leq Steps & l = l_3 \vee l = l_4 \\ true & c.c. \end{array} \right.$
- Si definimos, para un nodo l y un conjunto de relojes X la funcion de probabilidad $\mu_{X,l}$ tal que

  $$\mu_{X,l}(X', l') = \left\{ \begin{array}{ll} 1 & l' = l \wedge X' = X \\ 0 & c.c. \end{array} \right.$$

  tenemos que

$$\begin{array}{rll} prob & = \{ & (l_0, true, wake_{\mathcal{M},\mathcal{F}}, \mu_{\{x,y\},l_1}), \\ & & (l_1, true, sleep_{\mathcal{M},\mathcal{F}}, \mu_{\emptyset,l_0}), \\ & & (l_2, true, fail_{\mathcal{M},\mathcal{F}}, \mu_{\emptyset,l_5}), \\ & & (l_3, true, dontfail_{\mathcal{M},\mathcal{F}}, \mu_{\emptyset,l_4}), \\ & & (l_4, x = Step, \varepsilon, \mu_{\{x\},l_1}), \\ & & (l_4, true, sleep_{\mathcal{M},\mathcal{F}}, \mu_{\emptyset,l_0}), \\ & & (l_5, true, restorewake_{\mathcal{M},\mathcal{F}}, \mu_{\{x\},l_1}), \\ & & (l_5, true, restoresleep_{\mathcal{M},\mathcal{F}}, \mu_{\emptyset,l_0}) \ \} \\ & \cup \{ & (l_1, y < t_1, try, p) : p(\emptyset, l_2) = p_0 \wedge p(\emptyset, l_3) = 1 - p_0 \ \} \\ & \cup \{ & (l_1, t_i \leq y < t_{i+1}, try, p) : 1 \leq i \leq n \wedge p(\emptyset, l_2) = p_i \wedge p(\emptyset, l_3) = 1 - p_i \ \} \\ & \cup \{ & (l_1, t_n \leq y, try, p) : p(\emptyset, l_2) = p_n \wedge p(\emptyset, l_3) = 1 - p_n \ \} \end{array}$$

Para el caso en que la falla no tiene cambios de probabilidad, la definición es similar: se elimina el reloj y y disminye la cantidad de transiciones en *prob*.

Consideramos la falla $\mathcal{F}$ tal que:

$$Seq(Prob_{\mathcal{F}}) = (0, 0{,}01), (5, 0{,}05), (12, 0{,}06)$$

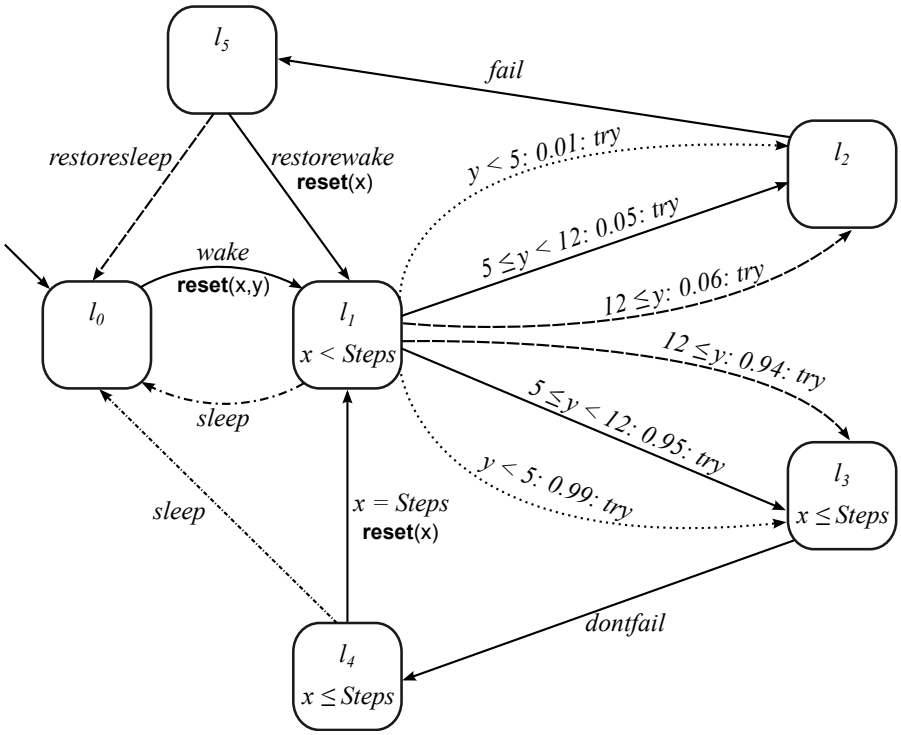


Figura 4.2: El PTA de la falla $\mathcal{F}$

es decir, cuando la falla pasa a estar habiliada, la probabilidad de ocurrencia comienza en 0,01. Luego de 5 unidades de tiempo, la probabilidad cambia a 0,05 y más tarde, luego de 12 unidades de tiempo, crece nuevamente a 0,06. En la figura 4.2 se puede ver el PTA que representa a esta falla.

4.3. Propiedades

Las propiedades a verificar en un sistema tolerante a fallas suelen diferir ligeramente de las que se verifican en un sistema que no lo es. En muchos casos, debemos tomar en cuenta la ocurrencia de fallas al momento de especificar la propiedad, dado que la tolerancia a fallas no siempre es completa. A veces, solo se pueden tolerar un número máximo de ocurrencias de una determinada falla, y otras veces la tolerancia es probabilística (es decir, el sistema tolera una ocurrencia de la falla con una probabilidad determinada).

Un ejemplo de este caso es la siguiente propiedad:

> *"Si el hospital se encuentra sin energía, entonces en a lo sumo 5 segundos el hospital volverá a tener energía"*

Esta propiedad no contempla un posible fallo en el generador de emergencia del hospital, pues si esto sucede, la energía no puede ser reestablecia

y por ende la propiedad no es cierta. Supongamos entonces que el hospital cuenta con 3 generadores de emergencia, para que en el caso de que surja un desperfecto en alguno de ellos, todavía pueda abastecerse. La propiedad anterior ahora estaría especificada de la siguiente forma:

> *"Si el hospital se encuentra sin energía y hay como máximo 2 generadores rotos, entonces en a lo sumo 5 segundos el hospital volverá a tener energía"*

Como puede verse, en el primer caso no se contempla la ocurrencia de fallas, mientras que en el segundo, si. Sin embargo, esto no significa que todas las propiedades deban cambiar cuando se modelan sistemas tolerantes a fallas.

4.3.1. Propiedades de Tolerancia

Además de lo dicho anteriormente, podemos distinguir un tipo característico de propiedad para los sistemas tolerantes a fallas, que denominaremos *propiedades de tolerancia*. Las mismas serviran para verificar si sistema tolera una determinada falla. Dado que nuestro modelo es probabilista y temporizado, también se pueden agregar aspectos relacionados a la probabilidad y al tiempo al momento de especificar una propiedad de tolerancia.

En nuestra lógica, una propiedad de tolerancia tiene la siguiente forma:

$$\mathcal{P}_{\geq 1}[\ \mathbf{G}\ (ocurre\_falla \rightarrow \mathcal{P}_{\geq p}[\ \mathbf{F}^{\leq t} sistema\_se\_recupera\ ])\ ]$$

Es decir, luego de la ocurrencia de la falla, el sistema volverá a un estado normativo en a lo sumo t unidades de tiempo con una probabilidad de por lo menos p.

Un ejemplo de este tipo de propiedades puede ser, volviendo a los ejemplos anteriores:

> *"Si un generador se rompe, entonces el hospital vuelve a tener luz en a lo sumo 5 segundos con una probabilidad mayor o igual a 0.95"*

La misma propiedad, especificada formalmente sería algo como:

$$\mathcal{P}_{\geq 1}[\ \mathbf{G}\ (generador\_se\_rompe \rightarrow \mathcal{P}_{\geq 0,95}[\ \mathbf{F}^{\leq 5} hospital\_tiene\_luz\ ])\ ]$$

Capítulo 5

OFFBEAT: Arquitectura y Lenguaje

En este capítulo describiremos el lenguaje y el funcionamiento de OFFBEAT, el model checker para sistemas tolerantes a fallas desarrollado. El lenguaje tiene una sintaxis muy similar a la de PRISM, pues los modelos son traducidos a modelos PRISM antes de ser verificados.

Hablaremos además de los problemas y limitaciones que se encontraron al momento de realizar el trabajo (y como se solucionaron), dado que la versión de PRISM con la que se trabajó es una versión en desarrollo.

5.1. Lenguaje

El lenguaje de la herramienta es una extensión del lenguaje de PRISM, descripto en la sección 2.6.2, con nuevas palabras claves para especificar fallas.

Recordemos que en nuestro modelo, las fallas constan de:

- Una *precondición*,
- Una función escalonada de *probabilidad*,
- Un conjunto de *efectos* sobre el modelo,
- Una *condición de restauración*, y
- Una cantidad de unidades de tiempo, que representan la *duración de cada intervalo* en donde la falla intentará ocurrir una vez.

Al describir una falla en nuestro lenguaje, habrá que especificar cada uno de estos elementos, a excepción el la duración del intervalo de tiempo, que será opcional y cuyo valor por defecto será 1. Esto ayuda a mejorar la abstracción, ya que el intervalo de tiempo no siempre es conocido por el ususario

de la herramienta, o a veces incluso no puede ser determinado con exactitud. En tal caso, el valor 1 para representar un intento de ocurrencia por cada unidad de tiempo parece la alternativa más razonable, pues representa el escenario en el cual, cuando la precondición se cumple, la falla puede ocurrir "en cualquier momento".

5.1.1. Fallas

Las fallas se declaran dentro de los módulos, después del invariante y antes de las transiciones del módulo, entre los constructores:

```
fault id_falla ... endfault
```

donde `id_falla` es un nombre único dentro del módulo. Dentro de estos constructores se especifican cada uno de los elemenotos de la falla.

Tanto la precondición de la falla como la condición de restauración serán expresiones booleanas sobre las variables privadas del módulo (aunque **no** sobre los relojes), describiendo cada una el conjunto de estados que le corresponde. La probabilidad estará dada por una probabilidad inicial, más una lista (posiblemente vacía) de cambios en el valor de la misma en instantes enteros de tiempo. El efecto de la falla se especificará de la misma forma que los cambios de estado en las transiciones, con un agregado para declarar el no-determinismo. Por último, el valor del intervalo sera simplemente un número entero, opcional.

```
fault FallaBizantina
  precondition: true;
  probability: 0.01, 2 -> 0.05, 6 -> 0.02;
  effect:   (x'=0) & (ok'=false)
        |   (x'=1) & (ok'=false)
        |   0.75: (x'=2) & (ok'=false) & (y'=0)
          + 0.25: (x'=2) & (ok'=false) & (y'=1);
  restores: ok;
  step: 3;
endfault
```

Figura 5.1: Un ejemplo de falla en nuestro lenguaje.

Entones, una posible especificación de una falla es la de la figura 5.1. El nombre de la misma es `FallaBizantina`. La precondición es `true`, lo que indica que la falla puede ocurrir en cualquier momento. La probabilidad inicial de ocurrencia es 0,01. Ésta probabilidad varía a 0,05 en el instante 2, y cambia nuevamente a 0,02 cuando pasan 6 unidades de tiempo. Cabe recordar que el tiempo para el cambio de probabilidades se comienza a contabilizar desde la última vez en que el sistema entró en un estado que

cumple la precondición de la falla desde un estado que no lo hace. En este caso en particular, dado que la precondición es `true`, este tiempo coincide con el tiempo desde el comienzo de la ejecución del sistema. El efecto tiene 3 opciones no deterministas (separadas con el caracter `|`), cada una de las cuales es una elección probabilística, especificada de la misma forma que en las transiciones normales del módulo. La condición de restauración es `ok`, es decir, el sistema estará restaurado de la falla cuando la variable `ok` sea igual a *true*. Finalmente, `step` indica la duración del intervalo de tiempo de la falla. En este caso, el valor es `3`, lo que indica que la falla intentará ocurrir una vez por cada intervalo de 3 unidades de tiempo.

El efecto de la falla no debería ser definido respecto del estado en la cual ésta ocurre. Esto significa que, si bien en la sección 4.1 la definición formal de una falla permite que el efecto de la misma sea especificado individualmente para cualquier estado, lo correcto sería que en el modelo del sistema no se haga referencia a un estado específico para definir el efecto de la falla, sino simplemente atenerse al conjunto de estados donde la misma puede ocurrir (definido en la precondición).

Además de declarar una falla, también puede especificarse que una transición es *deshabilitada* mientras una o más fallas están activas. La misma se hace usando la palabra clave `disabledby` al final de la transición, y especicando la lista de fallas que inhabilitan a la misma. Un posible ejemplo es el siguiente:

```
[] x>=1 & s=2 -> (s'=3) disabledby Falla1, Falla2, Falla3;
```

En el mismo se describe una transición que no puede ser tomada mientras cualquiera de las fallas `Falla1`, `Falla2`, `Falla3` está activa.

Veamos ahora el modelo en nuestro lenguaje del PTA ejemplo visto en las figuras 4.1a y 4.1b, con la falla ilustrada en 4.2 y considerando $Steps = 3$.

Notar que, si eliminamos la declaración de la falla, el modelo que nos queda es *exactamente* el de la figura 4.1a. Esto es bueno, ya que el objectivo es que las fallas puedan ser inyectadas al modelo sin necesidad de hacer cambios en las transiciones o variables del mismo.

5.1.2. Propiedades

Las propiedades en OFFBEATse especifican de la misma forma que en PRISM, pues el modelo que usamos es el mismo. Sin embargo, para hacerlo, contaremos con una variable "extra" que será de ayuda para verificar propiedades de sistemas tolerantes a fallas: la variable `active_m_f`, donde `f` será el nombre de la falla y `m` el del módulo al cual pertenece la falla. La misma es una condición que tendrá un valor de verdad `true` mientras la falla `f` esté activa, en el módulo `m`, y valor `false` en caso contrario. En el caso de la figura 5.2, se puede especificar la propiedad:

```
Pmax=? [ F<=5 active_Secuencia_FallaEjemplo & l=2 ]
```

```
module Secuencia
  l: [0..4] init 0;
  x: clock;

  invariant
    (true => x<=2)
  endinvariant

  fault FallaEjemplo
    precondition: (l>=1 & l<=2);
    probability: 0.01, 5 -> 0.05, 12 -> 0.06;
    effect:   (l'=1)
          |   0.8: (l'=l-1) & (x'=0)
            + 0.2: true;
    restores: (l>=2 & l<=3);
    step: 3;
  endfault

  [] (l=0 | l=2 | l=3) & x>2 -> (l'=l+1) & (x'=0);
  [] l=1 & x>2 -> 0.6: (l'=2) & (x'=0)
                + 0.4: (l'=3) & (x'=0);
  [] l=1 & x>=4 -> (x'=0) disabledby FallaEjemplo;
  [] l=4 & x>2 -> (x'=0);
endmodule
```

Figura 5.2: Modelo PRISM del PTA ejemplo en la figura 4.1a, aplicándole la falla de la figura 4.2

para calcular la máxima probabilidad de que la falla esté activa en un estado donde `l` sea igual a 2, en un instante de tiempo $t \leq 5$.

5.2. Traducción a PRISM

Como dijimos antes, el modelo descripto con el lenguaje de OFFBEAT debe ser traducido a un modelo en el lenguaje de PRISM. Para ello, transformaremos a cada módulo de nuestro modelo en un conjunto de módulos PRISM, que, al interactuar de manera síncrona, se comportarán de la manera esperada segun el modelo especificado en nuestra herramienta.

Los nuevos modelos en PRISM seran los correspondientes a los autómatas probabilistas temporizados $PTA'_{\mathcal{M}}$ y $PTA_{\mathcal{M},\mathcal{F}}$ para cada $\mathcal{F} \in F$, según lo descripto en la sección 4.2.

Formalmente, dado un modelo M de un módulo del sistema, con un conjunto F de fallas especificadas en el mismo, la traducción a un conjunto

de módulos PRISM consta de los siguientes pasos:

1. Considerar el módulo M' igual a M, pero sin las fallas.
2. Si F es no vacío, agregar a M' un reloj z. Este será el *reloj de fallas de M*.
3. Por cada $\mathcal{F} \in F$, agregar a M' una variable $flag_{\mathcal{F}}$, que indicará el estado en el que se encuentra la falla.
4. Modificar el invariante, de manera tal que se agreguen las condiciones necesarias para los nodos de salto, y para que los invariantes de M solo tengan efecto en estados "regulares", es decir, en estados que no sean nodos de salto.
5. Por cada falla $\mathcal{F} \in F$, agregar un conjunto de transiciones para sincronizar el comportamiento del módulo M' con el módulo correspondiente a $\mathcal{F}$.
6. A cada transición correspondiente a M, agregar en la guarda la condición de que el estado actual no sea un nodo de salto, y agregar también en cada efecto posible el reseteo del reloj de fallas z.
7. Por cada falla $\mathcal{F} \in F$, agregar un módulo $\mathcal{F}_M$, que modela parte del comportamiento de la falla y que se compone de manera síncrona con M'.

Cabe aclarar en este punto que, si bien en otros modelos de PRISM (por ejemplo MDP o CTMC) se puede desde un módulo hacer referencia a las variables de otros módulos en las guardas, esto no es así con los PTA. Se supone que en el futuro se agregará soporte para ésta funcionalidad. La misma hubiese sido de gran utilidad, ya que reduciría el número de transiciones agregadas a M', proveyendo una abstracción más clara del comportamiento de la falla con respecto al comportamiento del módulo principal.

Problemas con las acciones etiquetadas Otro problema que surgió durante el desarrollo de los modelos en PRISM fue el hecho de que las transiciones cuya guarda evalúa a $false$ son automáticamente eliminadas del modelo, *incluso* si tienen una etiqueta. Esto es un error de PRISM que se espera esté solucionado cuando se libere la versión final estable de la herramienta. Mientras tanto, esto nos perjudicaba, ya que el modelo PRISM es generado de manera automática, y no se verifican los valores de verdad de las guardas generadas antes de escribirlas al modelo. Por lo tanto, había casos en los cuales dos módulos se supone debían ejecutar una acción al mismo tiempo, o en el caso de que alguna de las guardas evaluara a $false$, ninguno debía ejecutarla en ningún momento. Sin embargo, al eliminar la transición

cuya guarda es falsa, el resultado es que el otro módulo puede ejecutar la acción sin restricciones de sincronización. La solución a este problema fue agregar a cada módulo un nodo de salto adicional, llamado *nodo dummy*, que actúa como estado inicial de cada módulo y que posee auto-ciclos sin restricciones de guarda o de relojes, etiquetados con cada una de las acciones de sincronización agregadas por OFFBEAT. De esta manera, el módulo poseerá al menos una transición con cada una de estas etiquetas, lo que forzará una sincronización incluso si otra transición con la misma etiqueta fue eliminada del modelo por tener una guarda falsa. El nodo dummy debe ser abandonado por todos los módulos al mismo tiempo, en el instante $t = 0$.

5.2.1. Ejemplo: Traducción del módulo principal

A modo de ejemplo, consideremos nuevamente el caso del PTA y su falla ilustrados en las figuras 4.1a y 4.2. El modelo en nuestra herramienta fue descripto en la sección anterior. Para este ejemplo supondremos que los nombres de variables `z`, `flag_f`, `xf`, `sf`, `reg_state`, `pre_f`, `rcond_f`, `jump_f`, `active_Secuencia_FallaEjemplo` y `dummy_m` no son usados por ningún módulo, como tampoco las etiquetas `wake`, `sleep`, `try`, `fail`, `dont_fail`, `restore_sleep`, `restore_wake` y `start`. En la traducción hecha por la herramienta estos nombres serán convenientemente elegidos para que sea muy poco probable que los mismos estén siendo usados en los módulos "originales".

El comienzo de la declaración del módulo principal es el mismo:

```
module Secuencia
  l: [0..4] init 0;
  x: clock;
```

En este punto se añaden las variables `dummy_m`, `flag_f` y el reloj `z`, de la siguiente forma:

```
  dummy_m: bool init true;
  flag_f: [0..3] init 0;
  z: clock;
```

La variable `dummy_m` es un booleano que indica si el módulo se encuentra en el nodo dummy o no. Su valor inicial, por lo tanto, debe ser `true`. `flag_f` es la encargada de indicar, dentro del módulo principal, el estado de la falla. Inicialmente su valor es `0` indicando que la falla comienza en estado inhabilitado. Finalmente, el reloj `z` será el reloj de fallas del módulo principal, usado para salir de los nodos de salto. En este punto cabe aclarar que sería conveniente contar la noción de *acciones urgentes*, haciendo referencias a transiciones que deben ser tomadas en el mismo instante en que se ingresa a un nodo desde el cual pueden ser tomadas. Esto nos permitiría deshacernos del reloj de fallas, simplificando el modelo. Actualmente las acciones urgenes pueden ser encontradas en el model checker Uppaal[6].

A continuación, corresponde la declaración de la condición invariante. La misma es distinta, ya que no solo debemos asegurar que no se pueda dejar pasar el tiempo en los nodos de salto sino que también debemos limitar el invariante original a los nodos en los cuales el sistema pueda comportarse de manera normativa (es decir, los estados que no son de salto). Para ello, ecribimos el invariante como sigue:

```
invariant
  (dummy_m => z<=0)
  & (jump_f => z<=0)
  & (reg_state => (true => x<=2))
endinvariant
```

Las primeras dos condiciones aseguran de que tanto el estado dummy como los estados de salto sean dejados en el mismo instante de tiempo en el que se llega a ellos. La tercera condición especifica que el invariante original solo debe ser satisfecho en los estados considerados regulares. Las fórmulas lógicas `jump_f` y `reg_state` serán definidas más adelante.

Luego, hay que describir las transiciones del módulo. Comenzaremos con las transiciones añadidas exclusivamente para evitar el problema de las acciones etiquetadas descripto anteriormente:

```
[wake] dummy_m -> true;
[sleep] dummy_m -> true;
[try] dummy_m -> true;
[fail] dummy_m -> true;
[dont_fail] dummy_m -> true;
[restore_sleep] dummy_m -> true;
[restore_wake] dummy_m -> true;

[start] dummy_m -> (dummy_m'=false);
```

Cada una de las primeras 7 transiciones representan auto-ciclos del nodo dummy, cada uno con una de las etiquetas de las transiciones del módulo de la falla. La última transición es la que se efectúa en el instante inicial para dejar el nodo dummy y comenzar con la ejecución normal. Todos los módulos del sistema tendrán una transición con etiqueta `start`.

Las siguientes transiciones son las que se añaden exclusivamente para lograr la sincronización con el módulo de la falla. Como se verá más adelante, cuando definamos la fórmula `jump_f`, todas estas transiciones (a excepción de la etiquetada con `try`) corresponden a nodos de salto. Las primeras dos,

```
[wake] pre_f & flag_f=0 -> (flag_f'=1);
[sleep] !pre_f & flag_f=1 -> (flag_f'=0);
```

contemplan los casos en los que la falla se habilita o deshabilita, de acuerdo al estado de la misma y al valor de verdad de la precondición en el nodo

actual. Estas acciones se corresponden con los casos (1) y (2) de la definición de $prob'$ en la sección 4.2.1.

La siguiente transición,

```
[try] pre_f & flag_f=1 -> (flag_f'=2) & (z'=0);
```

es la única que no corresponde a un nodo de salto. En la misma se refleja una posibilidad de que la falla ocurra. Para ello, la falla debe estar habilitada (`flag_f=1`) y la precondición debe cumplirse (`pre_f=true`). Si se toma esta transición, la falla pasa al estado en el cual debe "decidir" si efectivamente ocurre o no, y dado que este estado se corresponde con un nodo de salto, es necesario reiniciar el reloj de fallas `z`. Esta transición corresponde al caso (3) de la definición de $prob'$.

Para el caso en el que la falla no ocurra, se especifica la transición

```
[dont_fail] flag_f=2 -> (flag_f'=1);
```

donde la falla vuelve al estado de habilitada (caso (4) de la definición de $prob'$).

Ahora bien, si la falla ocurre, se debe tomar una de las siguientes transiciones:

```
[fail] flag_f=2 -> (l'=1) & (flag_f'=3);
[fail] flag_f=2 -> 0.8: (l'=l-1) & (x'=0) & (flag_f'=3)
                 + 0.2: (flag_f'=3);
```

Como puede verse, las mismas representan cada uno de los posibles efectos de la falla sobre las variables del módulo principal (caso (5) en la definición de $prob'$). En cada posible nuevo estado se setea convenientemente la variable `flag_f` a `3`, indicando que la falla se encuentra activa.

Finalmente, las transiciones

```
[restore_wake] flag_f=3 & rcond_f & pre_f -> (flag_f'=1);
[restore_sleep] flag_f=3 & rcond_f & !pre_f -> (flag_f'=0);
```

son tomadas cuando la falla es restaurada, pasando de estado activo a habilitada o inhabilitada, de acuerdo a si `pre_f` se cumple o no. Estas se corresponden con los casos (6) y (7) en la definición de $prob'$.

A continuación, debemos describir las transiciones del módulo original. Sin embargo, las mismas no seran exactamente iguales a las que se especificaron en el módulo, pues hay que recordar que éstas solo pueden ser tomadas en nodos que no sean de salto y además deben estar habilitadas. Más aún, en cada una de ellas hay que reinicar el reloj de fallas `z`, para asegurarnos que el valor del mismo al llegar a un nodo de salto es `0`. Por lo tanto, a cada transición se le agregarán una condición en la guarada y un reseteo de reloj:

```
[] reg_state & ((l=0 | l=2 | l=3) & x>2) ->
                                    (l'=l+1) & (x'=0) & (z'=0);
[] reg_state & (l=1 & x>2) -> 0.6: (l'=2) & (x'=0) & (z'=0)
                             + 0.4: (l'=3) & (x'=0) & (z'=0);
[] reg_state & !active_Secuencia_FallaEjemplo & (l=1 & x>=4) ->
                                                  (x'=0) & (z'=0);
[] reg_state & (l=4 & x>2) -> (x'=0) & (z'=0);
```

A todas las guardas se le agregó la condición `reg_state`, indicando que la transición solo puede ser tomada desde nodos que no sean nodos de salto. A la tercera transición, sin embargo, también se le agregó la condición `!active_Secuencia_FallaEjemplo`, pues la misma se deshabilita mientras la falla está activa. Además, a todos los posibles efectos de las transiciones se les añadió `(z'=0)`, para resetear el reloj de fallas. Estas transiciones corresponden al caso (8) en la definición de $prob'$ en la sección 4.2.1.

Con todas estas modificaciones, el módulo principal quedaría terminado como se mustra en la figura 5.3.

5.2.2. Ejemplo: Los módulos de fallas

Continuando con el ejemplo de la sección anterior, veremos ahora como se especifica el comportamiento del módulo de la falla, el cual, en sincronización con el módulo principal, completa la traducción del módulo. El mismo es una representación en PRISM del autómata probabilista temporizado que se muestra en la figura 4.2. Sin embargo, recordemos que debemos agregar el nodo dummy y la transición con etiqueta `start`, para evitar el problema descripto en la sección anterior.

En este punto es necesario hablar de una diferencia crucial en el uso de estos dos motores de verificación de PRISM. Como se explicó en la sección 2.6.2, la verificación usando el motor Digital Clocks es más eficiente, pero involucra una limitación en la especificación de restricciones de reloj: las mismas no deben ser estrictas. Esto es, restricciones como $x < 5$ o $x - y > 1$ no están permitidas. Esto representa un problema, pues como se puede ver en la sección 4.2.2, algunos invariantes y guardas deben necesariamente usar comparaciones estrictas. Por lo tanto, habrá algunas diferencias en la traducción cuando el modelo deba verificarse usando el motor de Abstraction Refinement con respecto a cuando se haga con Digital Clocks. Las mismas, junto con sus implicaciones, serán detalladas a lo largo de esta sección.

El módulo comienza entonces como cualquier otro:

```
module Secuencia_FallaEjemplo
```

El nombre del módulo involucra tanto el nombre de la falla como el nombre del módulo al que la misma pertence. Esto es así para permitir que distintos módulos tengan fallas independientes con el mismo nombre.

```
module Secuencia
  l: [0..4] init 0;
  x: clock;
  dummy_m: bool init true;
  flag_f: [0..3] init 0;
  z: clock;

  invariant
    (dummy_m => z<=0)
    & (jump_f => z<=0)
    & (reg_state => (true => x<=2))
  endinvariant

  [wake] dummy_m -> true;
  [sleep] dummy_m -> true;
  [try] dummy_m -> true;
  [fail] dummy_m -> true;
  [dont_fail] dummy_m -> true;
  [restore_sleep] dummy_m -> true;
  [restore_wake] dummy_m -> true;

  [start] dummy_m -> (dummy_m'=false);

  [wake] pre_f & flag_f=0 -> (flag_f'=1);
  [sleep] !pre_f & flag_f=1 -> (flag_f'=0);
  [try] pre_f & flag_f=1 -> (flag_f'=2) & (z'=0);
  [dont_fail] flag_f=2 -> (flag_f'=1);
  [fail] flag_f=2 -> (l'=1) & (flag_f'=3);
  [fail] flag_f=2 -> 0.8: (l'=l-1) & (x'=0) & (flag_f'=3)
                   + 0.2: (flag_f'=3);
  [restore_wake] flag_f=3 & rcond_f & pre_f -> (flag_f'=1);
  [restore_sleep] flag_f=3 & rcond_f & !pre_f -> (flag_f'=0);

  [] reg_state & ((l=0 | l=2 | l=3) & x>2) ->
                                    (l'=l+1) & (x'=0) & (z'=0);
  [] reg_state & (l=1 & x>2) -> 0.6: (l'=2) & (x'=0) & (z'=0)
                              + 0.4: (l'=3) & (x'=0) & (z'=0);
  [] reg_state & !active_Secuencia_FallaEjemplo & (l=1 & x>=4) ->
                                                  (x'=0) & (z'=0);
  [] reg_state & (l=4 & x>2) -> (x'=0) & (z'=0);
endmodule
```

Figura 5.3: El módulo principal del ejemplo, luego de la traducción a PRISM

A continuación se declaran las variables privadas y los relojes:

```
sf: [0..6] init 6;
xf: clock;
```

`sf` representa el conjunto de nodos de la falla. Notar que tiene un nodo más que en la figura 4.2: es el nodo dummy, representado por el valor `6`, que se especifica como valor inicial. Además, se declara el reloj `xf` propio de la falla.

En el caso de este ejemplo, sin embargo, se debe agregar un reloj adicional:

```
yf: clock;
```

que será necesario para contabilizar la duración de los intervalos de tiempo para los cuales la probabilidad cambia, según lo definido en el parámetro `probability`. Por lo tanto, se debe ser cauto al definir cambios de probabilidad para una falla ya que esto implica un reloj adicional en la misma, y un aumento más que significativo en la complejidad del modelo.

Luego se declara el invariante:

```
invariant
  (sf=1 => xf<3)
  & (sf=3 | sf=4 => xf<=3)
endinvariant
```

El mismo, en su conjunto, establece que la falla debe hacer un intento de ocurrencia una vez cada `3` unidades de tiempo. Este `3` proviene del valor del parámetro `step` en la declaración de la falla. Aquí ocurre la primera diferencia entre las traducciones correspondientes a los distintos motores de verificación. Como se puede ver, el invariante posee una declaración de una restricción estricta de reloj (`xf<3`), por lo que si se usa el motor Digital Clocks, la misma será inválida. La traducción para Digital Clocks simplemente cambia el `<` por `<=` . Las implicaciones que esto trae son mínimas: simplemente, se pueden dar dos intentos de ocurrencias en el mismo instante de tiempo (es decir, una de las ocurrencias en el instante final del intervalo de tiempo al cual pertenece, y la otra al comienzo del intervalo siguiente). Sin embargo, si consideramos un intervalo de tiempo $[N, M]$, con $N, M \in \mathbb{N}_0$ en el cual la falla puede ocurrir, la cantidad máxima y mínima de intentos de ocurrencia posibles es el mismo para ambos motores. Por lo tanto, esta diferencia en la traducción para ambos motores no debería tener implicancias en el resultado final.

A continuación comenzamos a declarar las transiciones del módulo de la falla. La primera de todas es

```
[start] sf=6 -> (sf'=0);
```

Como habíamos dicho antes, esta transición es efectuada por todos los módulos al iniciar la ejecución del sistema. Notar que, en este módulo, no hay un invariante relacionado a este nodo dummy, por lo cual, en principio, el módulo podría quedarse en este nodo cualquier cantidad de tiempo. Sin embargo, debemos recordar que esta acción se sincroniza con la correspondiente en el módulo principal, el cual sí tiene un invariante que obliga al módulo a dejar el nodo dummy en el instante 0. Por lo tanto, aunque no haya una condición invariante asociada al nodo dummy del módulo de la falla, el mismo es abandonado en el instante inical.

Seguidamente se declaran las transiciones

```
[wake] sf=0 -> (sf'=1) & (xf'=0) & (yf'=0);
[sleep] sf=1 | sf=4 -> (sf'=0);
```

En las mismas se refleja el cambio de estado de la falla de inhabilitada a habilitada y visceversa. Notar que, para que la falla pase a estar habilitada es necesario reiniciar los relojes `xf` y `yf`. Esto es así porque a partir del instante en que la falla se habilita, comienzan los intentos de ocurrencia y los cambios de probabilidad, y es necesario tener relojes "frescos" para regular estos intentos y cambios en su correspondiente intervalo de tiempo.

Luego se definen las transiciones correspondientes a los intentos de ocurrencia de la falla,

```
[try] sf=1 & yf<5 -> 0.01: (sf'=2)
                    + (1-0.01): (sf'=3);
[try] sf=1 & yf>=5 & yf<12: -> 0.05: (sf'=2)
                               + (1-0.05): (sf'=3);
[try] sf=1 & yf>=12: -> 0.06: (sf'=2)
                        + (1-0.06): (sf'=3);
```

Para cada "escalón" definido en el parámetro `probability` de la falla, se debe especificar una transición diferente, disjunta de las otras, donde se considere el intento de ocurrencia con la probabilidad establecida en el mismo, de acuerdo al valor de `yf`.

Aquí nos esncontramos con el segundo y peor problema de traducción para el caso del motor Digital Clocks. Para empezar, puede verse que las dos primeras transiciones poseen guardas donde se compara estrictamente el valor de un reloj con un valor entero. Ahora bien, supongamos que en estas comparaciones se reemplazan los `<` por `<=`. Este cambio afecta de forma muy significativa el comportamiento del módulo, pues por ejemplo en el instante donde `yf=5`, ahora hay que elegir no-determinísticamente entre las primeras dos transiciones declaradas, y por lo tanto la probabiliad de ocurrencia de la falla en ese momento depende de la elección que realice el adversario bajo el cual se esté actuando en ese momento. Esta es una limitacion que se presenta al trabajar con el motor de Digital Clocks.

Las dos transiciones siguientes,

```
[fail] sf=2 -> (sf'=5);
[dont_fail] sf=3 -> (sf'=4);
```

simplemente sincronizan con el módulo principal y diferencian en la falla el caso de que el intento de ocurrencia sea exitoso o no.

A continuación, se especifica una transición de "espera":

```
[] sf=4 & xf=3 -> (sf'=1) & (xf'=0);
```

La misma asegura que, en caso de que un intento de ocurrencia se de antes del final de su correspondiente intervalo, el próximo intento se dará solo después de que el presente intervalo termine. El módulo permanecerá en el estado representado por `sf=4` hasta que hayan pasado exactamente 3 unidades de tiempo (`xf=3`) desde el comienzo del intervalo.

Finalmente, las transiciones

```
[restore_wake] sf=5 -> (sf'=1) & (xf'=0) & (yf'=0);
[restore_sleep] sf=5 -> (sf'=0);
```

se tomarán cuando la falla se restaure: la primera en caso de que al momento de la restauración la precondición de la falla sea válida (por lo tanto, al igual que con la transición `sleep`, los relojes deben ser reseteados), y la segunda en caso contrario.

El módulo completo, finalmente, queda definido como se muestra en la figura 5.4.

5.2.3. Ejemplo: Constantes y Fórmulas

Además de los módulos, se agrega al modelo un conjunto de fórmulas que facilitan la comprensión del mismo. Una de estas fórmulas estará disponible para el usuario: `active_Secuencia_FallaEjemplo`. Las demás solo se usaran con el propósito de hacer el código más entendible y menos extenso. Por cada una de las fallas se declaran 4 fórmulas, y una más por cada módulo principal.

En el caso particular de este ejemplo, las fórmulas asociadas a la falla son

```
formula pre_f = (l>=1 & l<=2);
formula rcond_f = (l>=2 & l<=3);
formula active_Secuencia_FallaEjemplo = (flag_f=3);
formula jump_f = ( (pre_f & flag_f=0)
                 | (!pre_f & flag_f=1)
                 | (flag_f=2)
                 | (active_Secuencia_FallaEjemplo & rcond_f);
```

`pre_f` es la precondición de la falla, `rcond_f` es la condición de restauración, `active_Secuencia_FallaEjemplo` indica cuando la falla está activa,

```
module Secuencia_FallaEjemplo
  sf: [0..6] init 6;
  xf: clock;
  yf: clock;

  invariant
    (sf=1 => xf<3)
    & (sf=3 | sf=4 => xf<=3)
  endinvariant

  [start] sf=6 -> (sf'=0);
  [wake] sf=0 -> (sf'=1) & (xf'=0) & (yf'=0);
  [sleep] sf=1 | sf=4 -> (sf'=0);
  [try] sf=1 & yf<5 -> 0.01: (sf'=2)
                     + (1-0.01): (sf'=3);
  [try] sf=1 & yf>=5 & yf<12: -> 0.05: (sf'=2)
                               + (1-0.05): (sf'=3);
  [try] sf=1 & yf>=12: -> 0.06: (sf'=2)
                        + (1-0.06): (sf'=3);
  [fail] sf=2 -> (sf'=5);
  [dont_fail] sf=3 -> (sf'=4);
  [] sf=4 & xf=3 -> (sf'=1) & (xf'=0);
  [restore_wake] sf=5 -> (sf'=1) & (xf'=0) & (yf'=0);
  [restore_sleep] sf=5 -> (sf'=0);
endmodule
```

Figura 5.4: El módulo PRISM de la falla, generado por la traducción, usando el motor Abstraction Refinement

y `jump_f` es `true` si el módulo principal se encuentra en un nodo de salto relacionado a esta falla. Notar que los nombres de las fórmulas, salvo `active_Secuencia_FallaEjemplo`, fueron elegidos de manera que sean cortos y entendibles con el propósito de facilitar la comprensión del ejemplo, pero en la traducción, estos nombres serán más largos y con información de la falla y el módulo al cual están relacionados.

Tendremos además una fórmula correspondiente al módulo principal:

```
formula reg_state = !dummy_m & ( (!pre_f & flag_f=0)
                               | (pre_f & flag_f=1)
                               | (active_Secuencia_FallaEjemplo & !rcond_f));
```

la cual será verdadera cuando el módulo se encuentre en un estado que no sea de salto. Notar que en el caso de que el módulo estuviese asociado con más de una falla, la disjunción entre paréntesis deberá repetirse para todas

las fallas del módulo.

5.3. Implementación

En este trabajo se impementó el compilador que traduce modelos OFFBEAT a modelos PRISM. Para esto se usó el lenguaje Python[1] y la librería PLY[2] que es una implementación en Python de las conocidas librerías Lex y Yacc.

La arquitectura de OFFBEAT es de *tubos y filtros*: los componentes actúan en forma secuencial de manera tal que la salida de datos de un componente es la entrada del siguiente. En nuestro caso podemos distinguir 2 grandes componentes, el primero de los cuales se puede dividir en subcomponentes:

- **Compilador**: Toma el modelo realizado en el lenguaje de nuestra herramienta y lo traduce a un modelo en el lenguaje de PRISM. Este proceso es realizado secuencialmente por 3 sub-módulos:

  - *Lexer*, correspondiente al módulo Lex de la librería, posee un conjunto de expresiones regulares que definen como debe dividise el flujo de datos de entrada en *tokens* o partes. Aquí se declaran las palabras claves del lenguaje, los operadores, y la manera de formar literales tales como identificadores, números enteros o de punto flotante, etc.
  - *Parser*, que corresponde al módulo Yacc, define la manera en la que se forman las distintas construcciones sintácticas del lenguaje y el árbol sintáctico a partir de ellas. Cabe aclarar que las funciones para el parser provistas por PLY son muy flexibles, de manera que no es necesario armar el árbol sintáctico, sino que se pueden especificar acciones determinadas a realizar cuando se detectan las distintas construcciones. Sin embargo, en este trabajo se decidió usar la creación del árbol sintáctico (junto con algunos agregados), pues resultó la opción más apropiada para la traducción realizada por el compilador. Un dato importante es que el parser solo detecta que las construcciones estén bien formadas, aunque no se realiza ningún tipo de chequeo de tipos. Por lo tanto, este chequeo fue agregado manualmente en distintas partes del proceso de parsing.
  - *Traductor* toma la salida del Parser, en este caso un árbol sintáctico y un conjunto de fallas. En base a este árbol, escribe el módulo PRISM, con las modificaciones necesarias y el agregado de los módulos correspondientes a las fallas.

[1] http://www.python.org/
[2] http://www.dabeaz.com/ply/

- **PRISM**: Toma un modelo en lenguaje PRISM y una propiedad en PTCTL, y verifica que el modelo cumpla la propiedad especificada.

5.4. Limitaciones

En su versión actual, OFFBEAT presenta varias limitaciones, algunas relacionadas con las limitaciones propias de la versión de PRISM que se usó en el desarrollo del trabajo, y otras que son propias de la implementación en sí.

- Una de las grandes limitaciones es que, a diferencia de otros modelos como MDP o CTMC, no es posible hacer referencia en un módulo a las variables privadas de otro módulo.
- El uso de comandos PRISM está limitado a los que se explicaron en la sección 2.6.2, es decir, no se permite el uso de `system`, `reward`, `function`, entre otros.
- Las propiedades no pueden tener operadores `Pmax` o `Pmin` anidados, lo cual implica que no es posible especificar propiedades de tolerancia. Esta es una limitación impuesta por PRISM, y se espera que sea solucionada en futuras versiones.
- Los errores y mensajes devueltos por el compilador son exactamente los devueltos por PRISM, con lo cual puede haber referencias a variables o fórmulas creadas por el compilador y que no existen en el modelo original.
- En el proceso de compilación se omiten algunas verificaciones de tipos, aunque esos errores son detectados por PRISM. Asi mismo, el compilador no permite algunas declaraciones que si deberían permitirse, por ejemplo, el uso de constantes enteras en vez de literales enteros en algunas ocaciones.
- Si bien los nombres de variables, fórmulas, etiquetas y módulos elegidos por el compilador son lo suficientemente "ocultos", existe la posibilidad de que el usuario haga uso de alguno de estos nombres en su modelo, lo cual producirá un error.
- Cuando se usa el motor Digital Clocks, no se permite el uso de operadores de caminos acotados por tiempo en la especificación de propiedades. Sin embargo, en la mayoría de los casos, la cota de tiempo puede ser simulada mediante ligeros agregados al modelo y a la propiedad, lo cual veremos en la sección de casos de estudio.

Capítulo 6

Casos de Estudio

En éste capítulo aplicaremos las técnicas de model checking vistas para sistemas tolerantes a fallas, usando la herramienta desarrollada, a un conjunto de casos de estudio. En los mismos se describirán los módulos que componen cada sistema, el modelo de los mismos en el lenguaje de la herramienta, y un análisis de los resultados obtenidos, tomando en cuenta el rendimiento del model checker y las limitaciones impuestas por PRISM y por el tipo de modelo probabilístico usado.

Las pruebas fueron realizadas en una PC con procesador Intel Core 2 Duo T5550, con 2 GB de RAM y corriendo sobre Linux 32-bits con kernel 2.6.35-25-generic.

6.1. Sistema Europeo de Control de Trenes (ETCS)

En el próximo standard del Sistema Europeo de Control de Trenes[2], ETCS Level 3, se permite que trenes de alta velocidad viajen siguiéndose a poca distancia unos de otros. Los trenes se comunican regularmente con RBCs (Radio Block Centres) transmitiéndoles su posición exacta y su dirección, y esperando obtener permiso para avanzar (o "movement authorities", MA). Sin embargo, esta comunicación está basada en GSM-R, que es una adaptación del protocolo GSM y al tratarse de comunicación inalámbrica, es propensa a errores, lo que puede ocacionar retrasos en el envío y recepción de mensajes críticos[7].

El presente caso fue estudiado anteriormente en [7, 32, 17, 8]. En esta oportunidad se estudiará el efecto que ocasiona una posible pérdida de la comunicación entre un tren y el RBC.

6.1.1. Modelos en PTAs

Aqui presentaremos los modelos en términos de autómatas probabilistas temporizados de los distintos módulos que componen el sistema, que son los

trenes y los RBC.

Asumiremos el siguiente comportamiento: todos los trenes reportan su posición en intervalos de tiempos regulares. Asimismo, todos los trenes, con excepción del primero, esperan obtener, cada cierto tiempo, un permiso de avance del RBC más cercano. Si durante una determinada cantidad de tiempo el tren no recibe ningún MA, el mismo comenzará a frenar. Para que un RBC le conceda permiso de avance a un tren, debe haber recibido previamente información sobre la posición del tren que viene adelante.

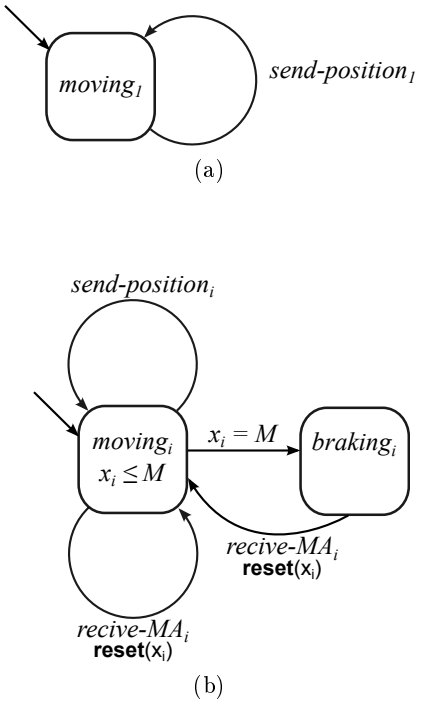


Figura 6.1: Los PTA correspondientes al primer tren y a cualquiera de los siguientes.

En la figura 6.1 se pueden ver los PTAs correspondientes tanto al primer tren como a un tren genérico. El primer tren (figura 6.1a) siempre puede avanzar, pues al no tener nigún tren por delante no hay peligro de choque. Por lo tanto, la única acción que ejecuta es la de enviar los datos sobre su posición al RBC, para que éste conceda permiso de avance al tren siguiente. Los demás trenes, cuyo comportamiento es representado por el autómata probabilista temporizado de la figura 6.1b, además de reportar su posición, deben esperar recibir MA's del RBC correspondiente y, en caso de no obtenerlo, iniciar maniobras de freno para evitar accidentes. Es por eso que en el PTA se puede ver que además de la acción *send-position*, tenemos la accion *recive-MA* y otra acción no etiquetada que representa el hecho de que, luego de M

unidades de tiempo, no se han recibido permisos de avance y por lo tanto se debe comenzar a frenar.

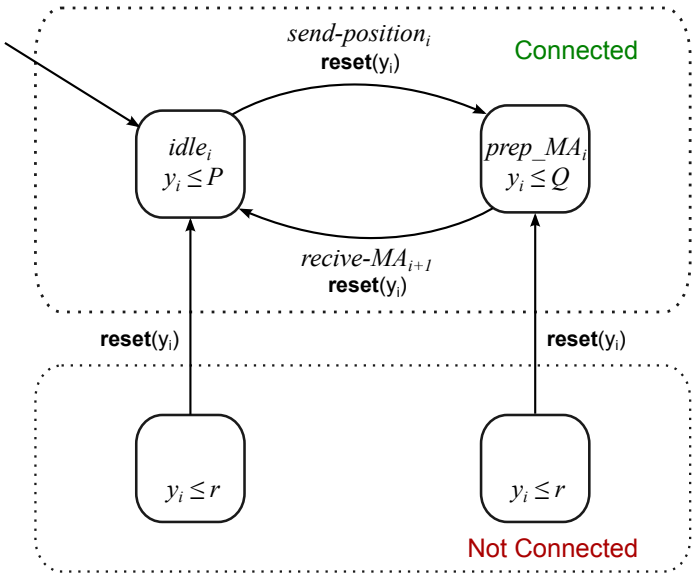


Figura 6.2: La representación del comportamiento de un RBC, considerando tanto los casos en los que hay conexión como los casos en que no.

En la figura 6.2 se muestra el comportamiento de un RBC. Cuando no hay problemas de conexión, el RBC recibe, luego P unidades de tiempo, los datos de la posicion del i-ésimo tren, y en base a esto, en Q unidades de tiempo envía una MA al $(i+1)$-ésimo tren. Cuando no ha conexión (los dos nodos de abajo) el RBC debe esperar una cantidad r variable de tiempo hasta que la conexión vuelva. Si bien estos nodos sin conexión no son nodos alcanzables en el PTA de la figura, lo serán cuando se inserte la falla correspondiente a la pérdida de conexión en el RBC.

6.1.2. Modelos en Offbeat

En esta sección veremos los modelos correspondientes a los autómatas probabilistas temporizados presentados en la sección anterior, expresados en el lenguaje de nuestro model checker. Consideraremos primero los modelos sin fallas, y luego agregaremos a los módulos de los RBC la posibilidad de ocurrencias de pérdidas de conexión, que implicarán que el RBC no podrá recibir datos de los trenes o mandar permisos de avance. Esto puede acarrear como consecuencia que algún tren comience a frenar sin que sea realmente necesario.

Comenzaremos definiendo algunas constantes:

```
const int POS_INTERVAL = 2;
```

```
const int MA_RECV_TIMEOUT=8;
const int MA_SENDING_TIME=2;
const int MAX_TIME_NO_CONN=5;
```

El significado y uso de las mismas quedará claro cuando se expliquen los modelos de trenes y RBCs.

El modelo para el primer tren es simple (figura 6.3). Notar que si bien la variable `braking_1` tiene 2 estados posibles, el estado donde su valor es `false` no es alcanzable. Por lo tanto, el modelo se corresponde con el PTA presentado aneriormente.

```
module train_1
    braking_1: bool init false;

    [pos_1] !braking_1 -> true;
endmodule
```

Figura 6.3: El modelo para el primer tren

Para el caso del modelo de un tren distinto del primero, representado en la figura 6.4, tenemos, además de la correspondiente variable `braking_i`, un reloj `toclock_i` que servirá para contabilizar el tiempo que ha transcurrido desde que se recibió la última MA. Notar que además de la transición para transmitir su posición, hay otras dos que representan la recepción de un MA y el comienzo de las maniobras de freno si no se recibe un MA en `MA_RECV_TIMEOUT` unidades de tiempo. Se puede ver además que, aún en el estado donde `braking_i` es `true`, es posible recibir un MA, y en ese caso, volver a avanzar con normalidad.

```
module train_i
    braking_i: bool init false;
    toclock_i: clock;

    invariant
        (!braking_i => toclock_i<=MA_RECV_TIMEOUT)
    endinvariant

    [pos_i] !braking_i -> true;
    [BRAKE_i] !braking_i & toclock_i>=MA_RECV_TIMEOUT ->
                                                (braking_i'=true);
    [MA_i] true -> (toclock_i'=0);
endmodule
```

Figura 6.4: El modelo para el i-ésimo tren

Por último, tenemos el módulo correspondiente al RBC (figura 6.5). Las variables en el mismo son `connected_i`, para indicar si hay o no conexión, y `st_i` para saber si el RBC está esperando por información sobre un tren (`st_i=0`) o preparando una MA para enviar (`st_i=1`). En este caso se omitieron las transiciones e invariantes correspondientes a los estados no alcanzables, es decir, aquellos que representan los nodos sin conexión, pués dado que todavía no se ha insertado la falla, no tiene sentido agregarlos.

El i-ésimo RBC representa al RBC que se encuentra entre los trenes i e $i+1$, es decir, recibe los datos de la posición del i-ésimo tren, y en base a esto concede permisos de avance al $(i+1)$-ésimo tren.

```
module rbc_i
    connected_i: bool init true;
    st_i: [0..1] init 0;
    rbcclock_i: clock;

    invariant
        (connected_i & st_i=0 => rbcclock_i<=POS_INTERVAL)
        & (connected_i & st_i=1 => rbcclock_i<=MA_SENDING_TIME)
    endinvariant

    [pos_i] st_i=0  & rbcclock_i>=POS_INTERVAL ->
                                (st_i'=1) & (rbcclock_i'=0);
    [MA_i+1] st_i=1 & rbcclock_i>=MA_SENDING_TIME ->
                                (st_i'=0) & (rbcclock_i'=0);
endmodule
```

Figura 6.5: El modelo para el i-ésimo RBC

Ahora bien, como dijimos antes, la falla que puede ocurrir en este módulo es una posible pérdida de la conexión. Esta falla puede ocurrir en cualquier momento siempre y cuando haya una conexión establecida, y se restaura automáticamente después de una cantidad variable de tiempo.

Para este caso, consideremos que la posibilidad de que se pierda la conexión en un determinado instante de tiempo es de `0.005`. Una posible especificación de la falla en el lenguaje de nuestra herramienta es la siguiente:

```
fault connection_loss_i
    precondition: connected_i;
    probability: 0.005;
    effect: 0.1: (connected_i'=false) & (rbcclock_i'=0)
          + 0.2: (connected_i'=false) & (rbcclock_i'=1)
          + 0.4: (connected_i'=false) & (rbcclock_i'=2)
          + 0.2: (connected_i'=false) & (rbcclock_i'=3)
```

```
          + 0.1: (connected_i'=false) & (rbcclock_i'=4);
    restores: connected_i;
endfault
```

Sin embargo, si solo agregamos la falla al modelo, tendremos como consecuencia que la conexión se perderá para siempre, pues no hay acciones salientes desde un estado donde `connected_i` sea igual a `false`. Por lo tanto, debemos añadir una transición que, luego de una cantidad determinada de tiempo sin conexión, traslade al módulo a un nodo donde `connected_i` sea verdadero. Además, debemos agregar un invariante que establezca que el valor del reloj `rbcclock_i` no puede superar un valor constante `MAX_TIME_NO_CONN`, y de esta manera hacer que cada posible efecto probabilístico represente una pérdida de conexión con distinta duración. Esta parte del modelo corresponde al tratamiento de la ocurrencia de la falla.

Agregamos entonces al invariante la cláusula:

```
!connected_i => rbcclock_i<=MAX_TIME_NO_CONN
```

y añadimos la transición

```
[] !connected_i & rbcclock_i>=MAX_TIME_NO_CONN ->
                              (connected_i'=true) & (rbcclock_i'=0);
```

También es necesario especificar de alguna forma que, mientras el módulo se encuentre sin conexión, no será posible enviar o recibir datos de los trenes. Sin embargo, eso se puede hacer facilmente usando la sentencia `disabledby` en las dos transiciones correspondientes a estas acciones.

Con estos cambios, el módulo se comporta de la manera que se espera. Notar que si, cuando ocurre la falla, la misma produce un efecto donde `rbcclock_i` es reseteado a un valor `t`, entonces el tiempo que dura la pérdida de conexión es `MAX_TIME_NO_CONN - t`. La versión final del módulo se muestra en la figura 6.6

6.1.3. Análisis de los resultados

El modelo sobre el que realizaremos la verificación consiste en 2 trenes y 1 RBC. Si bien cuando definimos los módulos solo se declaran dos relojes (`rbcclock_1` y `toclock_2`), el modelo final, luego de la traducción a PRISM, tendrá 2 relojes más: uno correspondiente al reloj agregado al módulo `rbc_1` y otro al módulo de la falla. Esto nos da un total de 4 relojes.

En este punto cabe mencionar que la verificación de un modelo con tal cantidad de relojes, usando el motor *Abstraction Refinement* resulta ineficiente. Por lo tanto, la verificación se efectuará usando el motor *Digital Clocks*.

La propiedad que nos interesará verificar en este caso es:

```
Pmax=? [ F<=t (braking_1 | braking_2)]
```

```
module rbc_i
    connected_i: bool init true;
    st_i: [0..1] init 0;
    rbcclock_i: clock;

    invariant
        (connected_i & st_i=0 => rbcclock_i<=POS_INTERVAL)
        & (connected_i & st_i=1 => rbcclock_i<=MA_SENDING_TIME)
        & (!connected_i => rbcclock_i<=MAX_TIME_NO_CONN)
    endinvariant

    fault connection_loss_i
        precondition: connected_i;
        probability: 0.005;
        effect: 0.1: (connected_i'=false) & (rbcclock_i'=0)
              + 0.2: (connected_i'=false) & (rbcclock_i'=1)
              + 0.4: (connected_i'=false) & (rbcclock_i'=2)
              + 0.2: (connected_i'=false) & (rbcclock_i'=3)
              + 0.1: (connected_i'=false) & (rbcclock_i'=4);
        restores: connected_i;
    endfault

    [] !connected_i & rbcclock_i>=MAX_TIME_NO_CONN ->
                            (connected_i'=true) & (rbcclock_i'=0);

    [pos_i] st_i=0  & rbcclock_i>=POS_INTERVAL ->
               (st_i'=1) & (rbcclock_i'=0) disabledby connection_loss_i;
    [MA_i+1] st_i=1 & rbcclock_i>=MA_SENDING_TIME ->
               (st_i'=0) & (rbcclock_i'=0) disabledby connection_loss_i;
endmodule
```

Figura 6.6: El modelo para el i-ésimo RBC, con la inclusión del código para fallas

es decir, la probabilidad máxima de que, en algún instante antes de `t` unidades de tiempo, alguno de los dos trenes se vea obligado a iniciar maniobras de freno. Se verificará la propiedad para distintos valores de `t`. El obejtivo de la verificación de esta propiedad será el de medir el desempeño de la herramienta.

Como se mencionó en la sección 5.4, la verificación de propiedades acotadas por tiempo no es soportada por el motor Digital Clocks. Por lo tanto, debemos emular este comportamiento en el modelo. La opción será agregar un módulo extra (que tenga un reloj) que se comporte de manera tal que, en un instante de tiempo determinado, setee un flag que pueda ser usado en la especificación de la propiedad para reemplazar la cota temporal. Sin embargo, debemos proceder con cuidado, pues al usar digital clocks, el proceso de verificación transforma los PTA en MDP, y en ese proceso se crea un nuevo estado **por cada posible valor entero de cada reloj**. Es decir que, al usar este motor de verificación, no solo el espacio se incrementará por la transformación de los relojes del modelo, sino también por la cota de tiempo que se desee especificar en la propiedad.

En nuestro caso, el valor máximo de `rbcclock_1` es el máximo entre `POS_INTERVAL`, `MA_SENDING_TIME` y `MAX_TIME_NO_CONN`, un número que es $O(10)$. El valor máximo de `toclock_2` es `MA_RECV_TIMEOUT`, que también es $O(10)$. Entonces, si queremos verificar la propiedad para `t=100`, tenemos que el espacio de estados se multiplica al menos por $10 * 10 * 100 = 10000$. Y además hay que tener en cuenta que la traducción a PRISM agrega relojes y estados al modelo. Por lo tanto, hay que ser muy cuidadoso con el tamaño del modelos y la cantidad y magnitud de los relojes, pues es bastante probable que se produzca una explosión de estados que desborde la capacidad de procesamiento de cualquier computadora ordinaria.

```
module prop
    prop_s: [0..1] init 0;
    prop_c: clock;
    invariant
        (prop_s=0 => prop_c<=PROP_TIME)
    endinvariant

    [] prop_s=0 & prop_c>=PROP_TIME -> (prop_s'=1);
endmodule
```

Figura 6.7: El módulo para la simulación de la cota de tiempo

Volviendo al caso de estudio, el módulo que agregaremos para simular la propiedad acotada por tiempo es el que se muestra en la figura 6.7. La variable `prop_s` tiene valor `0` solo durante las primeras `PROP_TIME` unidades de tiempo. Por lo tanto, ahora podemos especificar la propiedad como:

```
Pmax=? [ F prop_s=0 & (braking_1 | braking_2)]
```

Los resultados obtenidos para los distintos valores de t=PROP_TIME se muestran en la tabla 6.1.

t	Tamaño del modelo			Tiempo (seg)	Resultado
	Estados	Transiciones	Elecciones		
20	19147	45078	33760	3.402	0.007523
50	63067	143868	106420	10.475	0.019952
100	136267	308518	227520	27.793	0.042622
150	209467	473168	348620	58.128	0.063083
200	282667	637818	469720	97.627	0.084756
500	721867	1625718	1196320	585.824	0.200355
1000	1453867	3272218	2407320	1993.941	0.361495

Cuadro 6.1: Tabla de resultados para el caso de estudio de ETCS

Ahora verificaremos una propiedad de *safety*, que será equivalente a

"Los trenes nunca chocan."

Para que los trenes choquen, debería darse la siguiente situación:

1. El primer tren frena.
2. El segundo tren continua moviéndose por al menos una cantidad t de tiempo, por lo cual alcanza y colisiona con el primer tren.

Sin embargo, si observamos nuestro modelo, el primer tren no tiene una acción que represente la posibilidad de freno (pues al no tener ningún tren que lo preceda, la posibilidad de que frene depende de factores externos, por ejemplo desperfectos en la maquinaria, que están fuera del alcance de este análisis). Por lo tanto, para estudiar esta propiedad debemos agregar una transición que permita al tren pasar al estado en el que se encuentra efectuando maniobras de freno.

Para ello, agregaremos al primer tren las transiciones siguientes:

```
[BRAKE_1] !braking_1 -> (braking_1'=true);
[no_pos_1] braking_1 -> true;
```

La primera representa la posibilidad de que el tren frene en cualquier instante de tiempo. La segunda se agrega para sincronizar con el RBC de manera tal de evitar *timelocks* que surjan del hecho de que el RBC espera, cada POS_INTERVAL unidades de tiempo, un informe sobre la posición del tren (y dado que el mismo no puede ser entregado mientras el tren se encuentre realizando maniobras de freno). Por lo tanto también será necesario agregar al modelo del RBC la transición:

```
    [no_pos_1] st_1=0  & rbcclock_1>=POS_INTERVAL ->
                          (rbcclock_1'=0) disabledby connection_loss_1;
```

Para la verificación de la propiedad, modificaremos el módulo `prop` de manera tal que "avise" cuando pasen cierta cantidad de unidades de tiempo luego de que el primer tren frene. El módulo puede verse en la figura 6.8. Cuando el primer tren frena (`BRAKE_1`) se reinicia el reloj `prop_c` al mismo tiempo que se cambia el valor de `prop_s` a `1`. Luego de `PROP_TIME` unidades de tiempo, se vuelve a cambiar el valor de `prop_s` a `2`.

```
module prop
    prop_s: [0..2] init 0;
    prop_c: clock;
    invariant
        (prop_s=1 => prop_c<=PROP_TIME)
    endinvariant

    [BRAKE_1] prop_s=0 -> (prop_s'=1) & (prop_c'=0);
    [] prop_s=1 & prop_c=PROP_TIME -> (prop_s'=2);

endmodule
```

Figura 6.8: El módulo de la propiedad, modificado.

Por lo tanto, la propiedad de safety que nos interesará verificar será:

```
Pmax=? [ F prop_s=2 & !braking_2 ]
```

El resultado de la verificación de esta propiedad es una probabilidad máxima de 0,0, indicando que, efectivamente, los trenes no chocan y por lo tanto la propiedad de safety se satisface en el modelo.

La verificación de cualquiera de estas propiedades usando el motor Abstraction Refinement no fue viable, ya que incluso para cotas de tiempo pequeñas el análisis llegó a demorar más de 100 horas.

6.2. Bomba de Insulina Tolerante a Fallas (FTIP)

La terapia de bomba de insulina tolerante a fallas[9] (FTIP, Fault Tolerant Insulin Pump) está basada en la técnica de inyección subcutánea contínua de insulina, para personas con Diabetes. Esta enfermedad produce que la concentración de glucosa en la sangre sea más alta de lo normal, a causa de una deficiencia del páncreas al producir insulina, o de un nivel reducido en la efectividad de la insulina producida, o ámbos. En cualquier caso, una persona diabética debe "completar" el proceso de la producción de insulina manualmente.

El sistema de control FTIP hace uso de dos dispositivos, un *sensor* que mide el nivel de glucosa en la sangre a intervalos regulares de tiempo, y una *bomba* que inyecta insulina a través de una aguja muy fina insertada por debajo de la piel. La bomba posee un software que recibe los datos enviados por el sensor, y en base a éstos calcula la cantidad de insulina que se debe inyectar. La idea es que el proceso se realize de forma automática y cíclica, para mantener el nivel de glucosa en un rango seguro sin necesidad de que intervenga el paciente.

Cualquier falla o interrupción en el ciclo puede producir como resultado una *hiperglucemia* (alto nivel de glucosa) en el paciente. Por lo tanto, el hardware y el sowftware deben ser desarrollados con técnicas especiales que permitan lograr un comportamiento seguro y confiable.

Algunas de las fallas críticas que pueden ocurrir (y ser detectadas por el sistema) son:

- No se recibieron datos provenientes del sensor en una cantidad determinada de tiempo.
- El nivel de glucosa en sangre del paciente está fuera del rango seguro.
- La cantidad de insulina que debe inyectarse en el paciente no entra en el rango seguro programado.
- La insulina no está siendo correctamente inyectada por la bomba.

En principio, ante cualquiera de las anteriores situaciones, debería interrumpirse el proceso y avisar al paciente (a través de una alarma) de que una situación anormal está ocurriendo. Sin embargo, y dado que la interrupción del ciclo trae como consecuencia la intervención en el mismo por parte del paciente, se puede añadir redundancia y/o recuperación de errores para tolerar algunas de las condiciones críticas anteriores. Por ejemplo:

- Una posible razón por la que no se reciben datos del sensor es que el paciente puede estar rodeado de "ruido electrónico". Esta situación no es permanente, por lo tanto puede usarse un método numérico para estimar el nivel de glucosa en sangre de acuerdo a mediciones anteriores hasta que se vuelvan a recibir datos del sensor. Sin emabrgo, si la situación persiste por tiempo prolongado, deberá detenerse el ciclo e informar al paciente.
- Para evitar que la cantidad calculada de insulina a inyectar caiga fuera del rango seguro programado, pueden usarse varios algoritmos desarrollados independientemente unos de otros para realizar el mismo cálculo. Una vez obtenidos los distintos resultados, se efectúa una "votación" con la cual se pueden eliminar los resultados erróneos.

- Para detectar errores en el proceso de inyección de insulina, se usan dos sensores infrarrojos en distintas componentes de la bomba: uno monitorea el funcionamiento del motor, mientras que el otro hace lo propio con el movimiento del émbolo. Dado que estos dos componentes funcionan en conjunto, si realmente hay una falla, ámbos sensores deberían reportarla. Por lo tanto, y para evitar falsos positivos, si solo un sensor reporta un error el ciclo no se detiene.

En este caso estudiaremos la solución para el último item de la lista anterior: el chequeo del funcionamiento de la bomba a través de los dos sensores infrarrojos. La única falla posible es que el dispositivo deje de funcionar. Si esto ocurre, los dos sensores detectarán la falla. Sin embargo, aún cuando no se halla producido la falla, los sensores pueden llegar a reportar falsos positivos. El sistema dentendrá su ejecución solo si ambos dispositivos reportan un error.

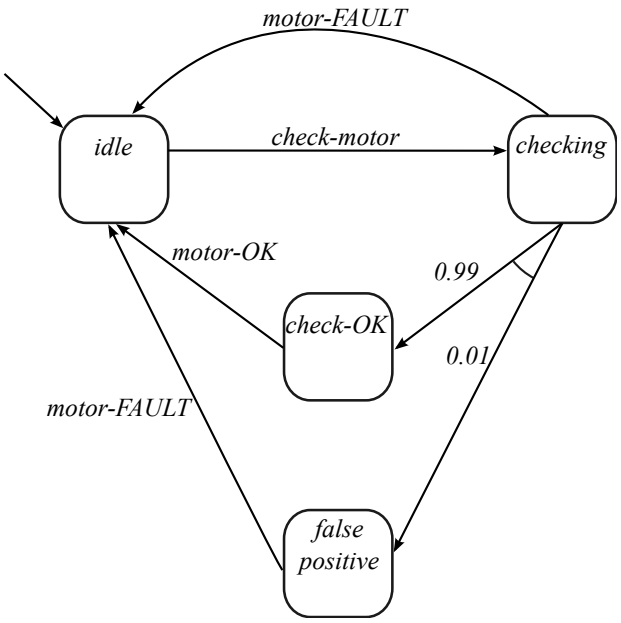


Figura 6.9: PTA del comportamiento del sensor infrarrojo del motor.

6.2.1. Modelo en PTAs

Para inyectar la insulina, la bomba se vale de un motor, que se pondrá en funcionamiento el tiempo que sea necesario para administrar la cantidad requerida. El movimiento del motor produce que un émbolo se desplace para así inyectar la insulina. El émbolo solo se desplazará a causa del movimiento del motor, a menos que el paciente intervenga manualmente.

La bomba cuenta con sensores infrarrojos para monitorear el correcto funcionamiento tanto del motor como del émbolo. Sin embargo, como se señaló en la sección anterior y dado que el motor y el émbolo funcionan en conjunto, solo se presentará una condición crítica cuando ambos sensores reporten una anomalía. Por lo tanto, el dispositivo solo detendrá su funcionamiento (avisando al paciente con una alarma) si ambos sensores detectan una falla. Sin embargo, existe la posibilidad de que incluso si el dispositivo funciona correctamente, algún sensor (o ambos) reporten un error. Esto se demonina un *falso positivo*.

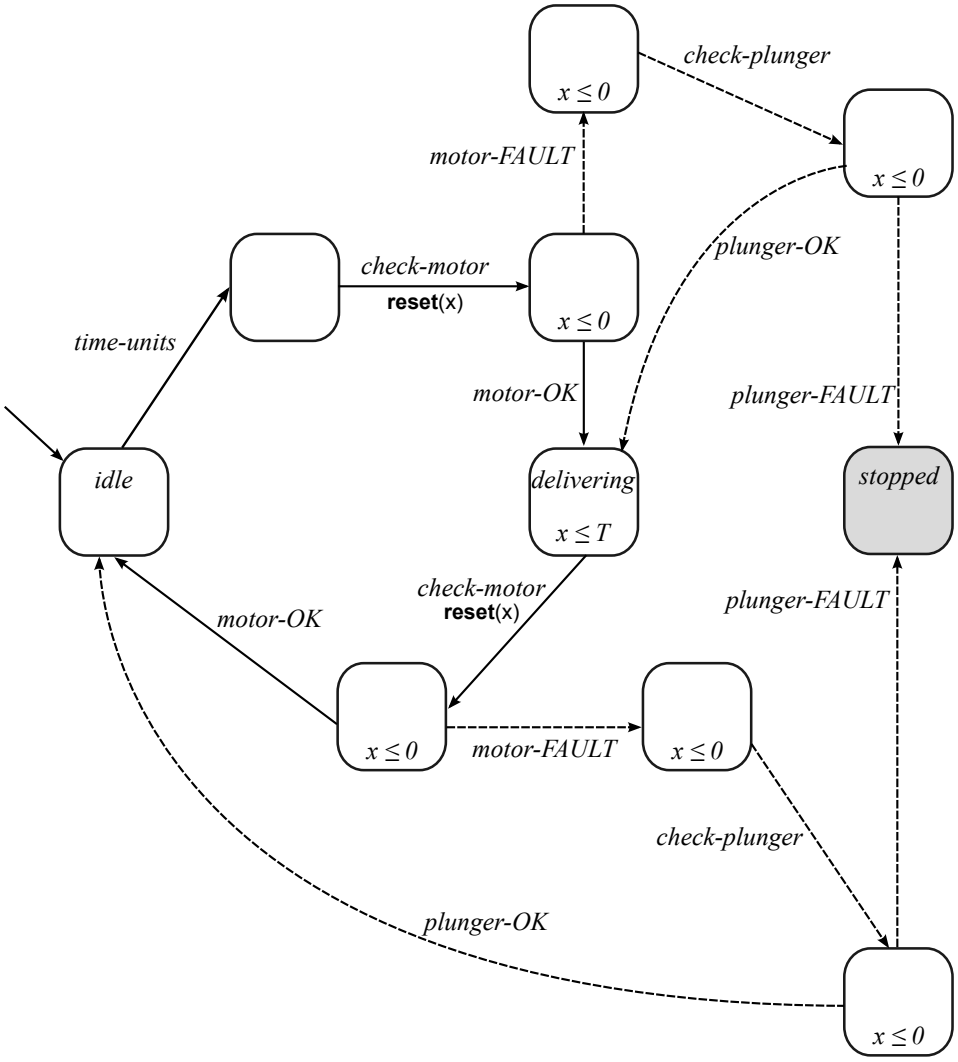


Figura 6.10: PTA del proceso principal.

El modelo (usando autómatas probabilistas temporizados) del dispositivo infrarrojo del motor se muestra en la figura 6.9. El dispositivo correspondien-

te al émbolo es análogo.

Cada sensor comienza en estado *idle* hasta que es necesario ejecutar un checkeo. Si el dispositivo no funciona, se avisará de la falla ejecutando la acción correspondiente (en el caso del sensor del motor, la acción *motorFAULT*. Sin embargo, aún si el dispositivo funciona correctamente, hay una pequeña posibilidad (en nuestro modelo, de 0.01) de que el sensor arroje un falso positivo.

El PTA del proceso encargado de manejar los dispositivos e inyectar la insulina puede verse en la figura 6.10.

El ciclo principal del proceso está dado por las flechas que no están dibujadas con guiones. El proceso se encuentra libre hasta que recibe (del proceso anterior) la cantidad de unidades de tiempo que debe trabajar el motor para entregar la cantidad de insulina requerida por el paciente. Entonces se efectúa el primer control de los dispositivos. Si el sensor del motor reporta que el mismo funciona correctamente, no es necesario controlar el otro sensor (recordemos que, para que el proceso se detenga, ambos sensores deben reportar una falla). Entonces se procede a la inyección de la insulina, haciendo funcionar el motor por T unidades de tiempo. Una vez terminado el suministro, se efectúa otra verificación del funcionamiento del dispositivo, antes de volver a quedar en estado libre. Si alguna verificación falla, entonces deberá controlarse el otro sensor (en el modelo, el sensor correspondiente al émbolo). Si éste sensor no reporta ninguna falla, se continúa la ejecución normalmente. Si en cambio el sensor también indica una falla, entonces el proceso debe detenerse, llevando al proceso al estado *stopped* (el nodo de color gris).

6.2.2. Modelo en Offbeat

Describiremos a continuación los modelos expresados en el lenguaje de Offbeat. Comenzaremos con el módulo correspondientes a los sensores (figura 6.11). Ámbos sensores se encuentran en el mismo módulo ya que una falla real en el motor o en el émbolo debería producir que ámbos dispositivos dejen de funcionar correctamente, pues los mismos están conectados.

Las variables `motor_ok` y `plunger_ok` son indicadores del correcto funcionamiento del motor y el émbolo, respectivamente. Las mismas comienzan teniendo el valor `true` y solo cambiarán a `false` si ocurre una falla en el dispositivo. Las otras variables, `md_s` y `pd_s` representan el estado interno de cada uno de los sensores, según el PTA de la figura 6.9. Las primeras 5 transiciones (desde `check_motor` hasta `motor_NOTOK`) son las transiciones correspondientes al sensor del motor, mientras que las otras son análogas y corresponden al sensor del émbolo. La falla `MotorFault` se especifica de manera tal que solo puede ocurrir mientras los dispositivos estén funcionando correctamente. La probabilidad de ocurrencia es muy baja, y el efecto es que tanto el motor como el émbolo dejen de funcionar. El sistema no pue-

```
module IRDetectors
    md_s: [0..3] init 0;
    pd_s: [0..3] init 0;
    motor_ok: bool init true;
    plunger_ok: bool init true;

    fault MotorFault
        precondition: motor_ok & plunger_ok;
        probability: 0.0005;
        effect: (motor_ok'=false) & (plunger_ok'=false);
        restores: false;
    endfault

    [check_motor] md_s=0 -> (md_s'=1);
    [motor_FAULT] md_s=1 & !motor_ok -> (md_s'=0);
    [] md_s=1 & motor_ok -> 0.99: (md_s'=2) // The motor is ok
                          + 0.01: (md_s'=3); // False positive
    [motor_OK] md_s=2 -> (md_s'=0);
    [motor_FAULT] md_s=3 -> (md_s'=0);

    [check_plunger] pd_s=0 -> (pd_s'=1);
    [plunger_FAULT] pd_s=1 & !plunger_ok -> (pd_s'=0);
    [] pd_s=1 & plunger_ok -> 0.99: (pd_s'=2) // The plunger is ok
                            + 0.01: (pd_s'=3); // False positive
    [plunger_OK] pd_s=2 -> (pd_s'=0);
    [plunger_FAULT] pd_s=3 -> (pd_s'=0);
endmodule
```

Figura 6.11: Código OFFBEAT para el modelo de los sensores

de restaurarse de la falla por sus propios medios, así que la condición de restauración es `false`.

El módulo del proceso principal emula el comportamiento del PTA que se mostró en la figura 6.10. Su descripción en OFFBEAT puede verse la figura 6.12.

La variable `s` representa los estados correspondientes al ciclo "regular" del proceso. Si se detecta una falla en el motor, el proceso deberá apartarse de ese ciclo regular para comprobar, mediante el otro sensor, el estado del émbolo. Éste alejamiento del funcionamiento normal se indicará seteando la variable `mf` a `true`. Si el émbolo se encuentra funcionando correctamente (`plunger_OK`) se retorna al ciclo regular de funcionamiento. En caso contrario, el sistema termina en el nodo *stopped*, ya que no puede continuar funcionando debido a la falla. El reloj `c` servirá para salir de los nodos de

salto, que en este caso serán los nodos correspondientes a los chequeos con los sensores, y además para asegurar que el proceso de inyección de la insulina dure DELIVERY_TIME unidades de tiempo.

```
module Delivering
    s: [0..4] init 0;
    t: [0..1] init 0;
    mf: bool init false;
    c: clock;
    stopped: bool init false;

    invariant
        (!stopped & (s=2 | s=4) => c<=0)
        & (!stopped & s=3 => c<=DELIVERY_TIME)
    endinvariant

    [time_units] !stopped & s=0 -> (s'=1);
    [check_motor] !stopped & s=1 -> (s'=2) & (c'=0);
    [motor_OK] !stopped & s=2 -> (s'=3) & (c'=0);
    [check_motor] !stopped & s=3 & c=DELIVERY_TIME ->
                                                (s'=4) & (c'=0);
    [motor_OK] !stopped & s=4 -> (s'=0);

    [motor_FAULT] !stopped & (s=2 | s=4) ->
                                          (mf'=true) & (t'=0);
    [check_plunger] !stopped & mf & t=0 -> (t'=1);
    [plunger_OK] !stopped & mf & t=1 & s=2 ->
                        (mf'=false) & (t'=0) & (s'=3) & (c'=0);
    [plunger_OK] !stopped & mf & t=1 & s=4 ->
                                 (mf'=false) & (t'=0) & (s'=0);

    [plunger_FAULT] !stopped & mf & t=1 -> (stopped'=true);
endmodule
```

Figura 6.12: Código OFFBEAT para el proceso *delivering*.

6.2.3. Análisis de los resultados

Para probar el modelo especificado en la sección anterior, verificaremos sobre el mismo la propiedad

```
Pmax=? [F<=t stopped & !active_IRDetectors_MotorFault]
```

es decir, la probabilidad de que el proceso se detenga sin que halla ocurrido una falla en el dispositivo (o sea, debido a los falsos positivos). La verificación

se realizará sobre distintos valores para t.

Para esto será necesario agregar, además de los dos módulos de la sección anterior, un módulo extra para lidiar con las restricciones de tiempo en la especificación de la propiedad, tal y como hicimos con el caso de estudio anterior, ya que el motor Digital Clocks no nos permite verificar propiedades acotadas por tiempo. El módulo agregado será similar al del caso anterior.

Los resultados obtenidos pueden verse en la tabla 6.2.

t	Tamaño del modelo			Tiempo (seg)	Resultado
	Estados	Transiciones	Elecciones		
100	59283	99597	85732	7.677	0.003991
200	119683	200897	172932	36.951	0.007676
500	300883	504797	434532	320.861	0.017612
1000	602883	1011297	870532	1246.679	0.030981

Cuadro 6.2: Tabla de resultados para el caso de estudio de la FTIP

Además de la propiedad anterior, nos interesa verificar la propiedad de safety

> *"Si se produce una falla en el dispositivo, entonces luego de T unidades tiempo no se intentará inyectar insulina"*

Para hacerlo debemos agregar, al igual que en el caso anterior, un módulo que nos sirva de soporte para el modelado de la propiedad, debido a las limitaciones de la herramienta para especificar propiedades. El módulo será el que se muestra en la figura 6.13. Notar que fue necesario usar la etiqueta `__fail_IRDetectors_MotorFault__`, correspondiente a la traducción a PRISM del modelo, para detectar la ocurrencia de la falla.

```
module prop
    prop_f: [0..2] init 0;
    prop_c: clock;

    invariant
        (prop_f=1 => prop_c<=DELIVERY_TIME+1)
    endinvariant

    [__fail_IRDetectors_MotorFault__] prop_f=0 -> (prop_f'=1) & (prop_c'=0);
    [] prop_f=1 & prop_c=DELIVERY_TIME+1 -> (prop_f'=2);
endmodule
```

Figura 6.13: Módulo de soporte para la propiedad de safety en el caso de la FTIP

En este caso, $T =$ `DELIVERY_TIME+1`. La especificación de la propiedad en PTCTL es:

```
Pmax=? [F prop_f=2 & s=3]
```

pues recordemos que el estado `s=3` corresponde a la inyección de la insulina.

El resultado del cálculo de esta probabilidad máxima es de $0,0$, con lo cual podemos garantizar que la propiedad se cumple.

Al igual que en el caso de estudio anterior, no fue posible la verificación de ninguna de estas propiedades usando el motor Abstraction Refinement, pues la misma toma demasiado tiempo debido al alto número de relojes en el modelo final.

Capítulo 7

Conclusión

El objetivo de este trabajo fue desarrollar una herramienta de model checking que permita un modelado ordenado e intuitivo de sistemas tolerantes a fallas, distinguiendo el comportamiento normativo del sistema de la ocurrencia de fallas. Esto se logró extendiendo el lenguaje del model checker PRISM, agregando una estructura para declarar fallas, especificando algunos aspectos relevantes de la misma, tales como probabilidad de ocurrencia y el effecto que tienen sobre el comportamiento del sistema. Al trabajar sobre la versión más reciente de PRISM, el model chequer tiene soporte para modelos probabilísticos temporizados.

Una de las dificultades con las que nos encontramos durante el desarrollo de la herramienta fue el hecho de que la versión de PRISM con la que trabajamos no era una versión estable. Esto implicó que en muchos casos surgieron errores o comportamientos no previstos que eran causados por bugs del model checker. Debemos agradecer en este contexto a David Parker, que amablemente recibió los reportes de errores que le enviamos y nos guió en como resolverlos.

Otro problema tuvo que ver con la complejidad de los modelos generados. Como se explicó, la incorporación de fallas implica la adición de más de un reloj al modelo, lo que aumenta fuertemente el tamaño del mismo y la complejidad de la verificación. En algunos casos, nos vimos obligados a reducir de manera drástica algunos modelos para poder efectuar la verificación. Esta complejidad en los modelos finales fue también el causante de que no sea posible usar el motor Abstraction Refinement para la verificación de propiedades, ya que en la gran mayoría de los casos este proceso duraba demasiadas horas y consumía una gran cantidad de recursos del ordenador (para el caso de estudio del Sistema de Control de Trenes Europeo, la verificación llegó a consumir más de 10 GB de memoria RAM).

Sin embargo, podemos decir que el desarrollo de la herramienta cumplió con los objetivos planteados al inicio del trabajo, ya que mediante un lenguaje sencillo se permite modelar sistemas con fallas, abstrayendo estas últimas del

comportamiento del sistema.

Trabajo a futuro

Entre los posibles objetivos a futuro en el desarrollo de esta herramienta, podemos considerar el hecho de disminuir la cantidad de relojes usados. Para esto sería util esperar a que PRISM implemente la posibilidad de hacer referencia a variables en módulos externos para PTAs. Sería también de ayuda la incorporación de la noción de "acciones urgentes", como de las que dispone el model checker Uppaal. Ésto permitiría construir modelos más ricos que traduzcan en estructuras con una complejidad manejable.

El concepto de *propiedades de tolerancia*, introducido en la sección 4.3.1, sería muy útil en la verificación de sistemas tolerantes a fallas. Lamentablemente, en este caso también será necesario esperar a que PRISM agregue soporte para operadores $\mathcal{P}_{\bowtie p}[\ ]$ anidados.

Sería también conveniente agregar soporte para el lenguaje completo de PRISM, los que permitiría usar sentencias como `reward`, `function` y `system` para describir sistemas más complejos o verificar nuevos tipos de propiedades.

Apéndice A

Manual de OFFBEAT

En el presente apéndice se indica como preparar la herramienta para efectuar verificaciones, y cuales son algunas de las opciones con las que la misma cuenta. Además, se presentaran algunos ejemplos de ejecución.

Pasos previos y utilización de OFFBEAT

En necesario tener instalado Python, versión 2.6 o superior.

Primero se deben generar los scripts ejecutables de PRISM. Para eso, entrar desde una terminal al directorio `prism4.0.beta.r2410-src` situado en el interior del directorio de OFFBEAT y ejecutar el comando `make`.

Para usar OFFBEAT en la verificación de modelos, se deberá ejecutar el archivo `offbeat.py` desde la consola, indicando a continuación los archivos que corresponden al modelo y a las propiedades, en ese orden. Por ejemplo, si se quiere verificar un modelo guardado en un archivo `model.nm`, contra un conjunto de propiedades especificadas en `prop.pctl`, el comando a ejecutar (suponiento que nos encontramos en el directorio de la herramienta) es:

```
nico@halibel:~/offbeat$ ./offbeat.py model.nm prop.pctl
```

Algunas opciones

Listaremos a continuación algunas de las opciones disponibles para la herramienta. Algunas son propias de OFFBEAT, mientras que otras lo son de PRISM. Sin embargo, en algunos casos, OFFBEAT utiliza algunas opciones de PRISM para generar cambios en los modelos generados.

- **-ptamethod <motor_de_verificación>**: Opción para indicar el motor de verificación que se desea usar. Las opciones son **games**, correspondiente al motor Abstraction Refinement (opción por defecto), y **digital**, para el motor Digital Clocks.

- **-prismfn <archivo>**: Nombre del archivo que contendrá el modelo intermedio, es decir, la traducción a PRISM del modelo especificado. Por defecto el nombre de este archivo es `compiler-result.nm`.
- **-fixdl**: Opción de PRISM para añadir auto-ciclos a los estados que no tienen transiciones salientes, para evitar deadlocks.
- **-exportresults <archivo>**: Opción de PRISM para exportar los resultados de la verificación a un archivo

Ejemplos

Presentamos a continuación algunos ejemplos de ejecución con distintas opciones.

En el primer caso,

```
$> ./offbeat.py model.nm prop.pctl -prismfn model.prism -fixdl
```

la verificación se realiza usando el motor Abstraction Refinement. El modelo intermedio se escribe en el archivo `model.prism`, y además, se añaden auto-ciclos a los estados que no poseen transiciones de salida, para así evitar deadlocks.

En el siguiente caso:

```
$> ./offbeat.py model.nm prop.pctl -ptamethod digital
```

La verificación se realiza usando el motor Digital Clocks, y el modelo intermedio se guarda en el archivo `compiler-result.nm`.

Finalmente, con el comando

```
$> ./offbeat.py model.nm prop.pctl -exportresults out.offbeat
-prismfn model.tmp
```

se efectúa la verificación usando el motor Abstraction Refinement, el modelo PRISM se guarda en el archivo `model.tmp`, y los resultados de esta verificación son guardados en `out.offbeat`.

Apéndice B

Sintaxis Formal de OFFBEAT

A continuación se describirá la sintaxis de OFFBEAT como una gramática libre de contexto, expresada en BNF. No serán tomados en cuenta los espacios en blanco, tabulaciones o saltos de línea. Con `null` nos referiremos a la producción vacía.

```
    <MODEL>  ::=  <MODULE> <MODEL>
               |  <CONSTANT> <MODEL>
               |  <FORMULA> <MODEL>
               |  <MODULE>

<CONSTANT>   ::=  "const" <TYPE> <ID> = <EXPRESSION> ";"
    <TYPE>   ::=  "bool" | "int" | "double"

 <FORMULA>   ::=  "formula" <ID> = <EXPRESSION> ";"

  <MODULE>   ::=  module <ID> <LOCAL_VARIABLES> <INVARIANT> <FAULT_SET>
                  <TRANSITION_SET> "endmodule"
```

Un modelo consta de uno o más módulos, y alguna cantidad finita de constantes y fórmulas. La producción `<ID>` hace referencia a cadenas alfanuméricas (pueden contener el caracter _) que puedan ser usadas como nombres de variables (o de módulos) y que no sean palabras reservadas. Las producciones `<INTVALUE>` y `<DOUBLEVALUE>` corresponden a cadenas numéricas que correspondan a valores enteros o de punto flotante, respectivamente.

Veamos ahora los componentes de un módulo. En primer lugar, la declaración de variables locales:

```
<LOCAL_VARIABLES>  ::=  <VAR_DECLARATION> <LOCAL_VARIABLES>
                     |  null
<VAR_DECLARATION>  ::=  <ID> ":" <RANGE> "init" <INTVALUE> ";"
                     |  <ID> ":" <RANGE> "init" <ID> ";"
                     |  <ID> ":" "bool" "init" "true" ";"
                     |  <ID> ":" "bool" "init" "false" ";"
                     |  <ID> ":" "bool" "init" <ID> ";"
                     |  <ID> ":" "clock" ";"
          <RANGE>  ::=  "[" <INTVALUE> ".." <INTVALUE> "]"
                     |  "[" <INTVALUE> ".." <ID> "]"
                     |  "[" <ID> ".." <INTVALUE> "]"
                     |  "[" <ID> ".." <ID> "]"
```

Luego, la declaración del invariante:

```
<INVARIANT>  ::=  "invariant" <EXPRESSION> "endinvariant"
               |  null
```

A continuación, una lista de una o más fallas:

```
<FAULT_SET>  ::=  <FAULT> <FAULT_SET>
               |  null

    <FAULT>  ::=  "fault" <ID> <PRECONDITION> <PROBABILITY>
                  <EFFECT> <RESTORES> "endfault"
               |  "fault" <ID> <PRECONDITION> <PROBABILITY>
                  <EFFECT> <RESTORES> <STEP> "endfault"
```

Los componentes de una falla son: precondición, probabilidad de ocurrencia, effecto que produce, condición de restauración, y, como parámetro opcional, duración del intervalo en el cual puede ocurrir la falla. Detallamos la sintaxis de estos componentes a continuación:

```
<PRECONDITION>  ::=  "precondition" ":" <EXPRESSION> ";"

 <PROBABILITY>  ::=  "probability" ":" <NUMBER> ";"
                  |  "probability" ":" <NUMBER> <PROB_CHANGES> ";"
<PROB_CHANGES>  ::=  <PROB_CHANGES> <PROB_CHANGES>
                  |  "," <INTVALUE> "->" <NUMBER>

      <EFFECT>  ::=  "effect" ":" <EFFECT_LIST> ";"
 <EFFECT_LIST>  ::=  <EFFECT_LIST> "|" <EFFECT_LIST>
                  |  <PROB_UPDATE>

    <RESTORES>  ::=  "restores" ":" <EXPRESSION> ";"

        <STEP>  ::=  "step" ":" <INTVALUE> ";"
```

A continuación se deben declarar las transiciones del modelo. Las producciones correspondientes a las mismas se detallan a continuación:

```
<TRANSITION_SET>  ::=  <TRANSITION> <TRANSITION_SET>
                     |  null
    <TRANSITION>  ::=  "[" <ID> "]" <EXPRESSION> "->" <PROB_UPDATE> ";"
                     |  "[" <ID> "]" <EXPRESSION> "->" <PROB_UPDATE>
                        "disabledby" <FAULT_ID_LIST> ";"
                     |  "[" "]" <EXPRESSION> "->" <PROB_UPDATE> ";"
                     |  "[" "]" <EXPRESSION> "->" <PROB_UPDATE>
                        "disabledby" <FAULT_ID_LIST> ";"
 <FAULT_ID_LIST>  ::=  <ID>
                     |  <FAULT_ID_LIST> "," <FAULT_ID_LIST>
   <PROB_UPDATE>  ::=  <PROB_UPDATE> "+" <PROB_UPDATE>
                     |  <EXPRESSION> ":" <UPDATE>
                     |  <UPDATE>
        <UPDATE>  ::=  <UPDATE> "&" <UPDATE>
                     |  "(" <ID> "'=" <EXPRESSION> ")"
                     |  true
```

Finalmente, las producciones correspondientes a las expresiones estan divididas en varios casos. Notar que no se realiza distinción alguna entre expresión aritmética y expresión booleana. Éste chequeo lo lleva a cabo el compilador luego de analizar la sintaxis.

```
<EXPRESSION>  ::=  <EXPRESSION> "+" <EXPRESSION>
                |  <EXPRESSION> "-" <EXPRESSION>
                |  <EXPRESSION> "*" <EXPRESSION>
                |  <EXPRESSION> "/" <EXPRESSION>
                |  <EXPRESSION> "<" <EXPRESSION>
                |  <EXPRESSION> "<=" <EXPRESSION>
                |  <EXPRESSION> ">" <EXPRESSION>
                |  <EXPRESSION> ">=" <EXPRESSION>
                |  <EXPRESSION> "=" <EXPRESSION>
                |  <EXPRESSION> "!=" <EXPRESSION>
                |  <EXPRESSION> "&" <EXPRESSION>
                |  <EXPRESSION> "|" <EXPRESSION>
                |  <EXPRESSION> "=>" <EXPRESSION>
                |  "!" <EXPRESSION>
                |  "(" <EXPRESSION> ")"
                |  <NUMBER>
                |  <BOOLEAN>
                |  <ID>
  <BOOLEAN>   ::=  true
                |  false
   <NUMBER>   ::=  <INTVALUE>
                |  <DOUBLEVALUE>
```

Las palabras reservadas del lenguaje se listan en la siguiente producción:

```
<KEYWORD>  ::=  "const"        |  "formula"
             |  "module"       |  "endmodule"
             |  "clock"        |  "bool"
             |  "int"          |  "double"
             |  "invariant"    |  "endinvariant"
             |  "init"         |  "fault"
             |  "endfault"     |  "precondition"
             |  "probability"  |  "effect"
             |  "restores"     |  "step"
             |  "disabledby"   |  "true"
             |  "false"
```

Apéndice C

Ejemplo de traducción a PRISM

En este apéndice se muestra el código PRISM generado para el caso de estudio del Sistema de Control de Trenes Europeo. En el mismo se muestran los nombres de variables y las etiquetas de las transiciones tal y como son generados por el traductor, ya que en el ejemplo de la sección 5.2 se eligieron estos nombres de manera tal que sean legibles y ayuden a la comprensión de los casos.

```
pta
const int POS_INTERVAL = 2;
const int MA_RECV_TIMEOUT = 8;
const int MA_SENDING_TIME = 2;
const int MAX_TIME_NO_CONN = 5;
const int PROP_TIME = 20;

module prop

    prop_s: [0..1] init 0;
    prop_c: clock;

    invariant
        (prop_s=0=>prop_c<=PROP_TIME)
    endinvariant

    [] prop_s=0&prop_c=PROP_TIME -> (prop_s'=1);

endmodule

module train_1

    braking_1: bool init false;

    [pos_1] !braking_1 -> true;
```

```
endmodule

module rbc_1

    connected_1: bool init true;
    st_1: [0..1] init 0;
    rbcclock_1: clock;
    __rbc_1_dummy: bool init true;
    __rbc_1_z: clock;
    __flag_rbc_1_connection_loss_1: [0..3] init 0;

    invariant
        (__rbc_1_dummy => __rbc_1_z<=0) & (__jump_rbc_1_connection_loss_1 => __rbc_1_z<=0)
        & (__rbc_1_reg_state => ((connected_1&st_1=0=>rbcclock_1<=POS_INTERVAL)
                                &(connected_1&st_1=1=>rbcclock_1<=MA_SENDING_TIME)
                                &(!connected_1=>rbcclock_1<=MAX_TIME_NO_CONN)))
    endinvariant

    [__start__] __rbc_1_dummy -> (__rbc_1_dummy'=false);
    [__restore_wake_rbc_1_connection_loss_1__] __rbc_1_dummy -> true;
    [__restore_sleep_rbc_1_connection_loss_1__] __rbc_1_dummy -> true;
    [__wake_rbc_1_connection_loss_1__] __rbc_1_dummy -> true;
    [__sleep_rbc_1_connection_loss_1__] __rbc_1_dummy -> true;
    [__try_rbc_1_connection_loss_1__] __rbc_1_dummy -> true;
    [__dontfail_rbc_1_connection_loss_1__] __rbc_1_dummy -> true;
```

```
[__fail_rbc_1_connection_loss_1__] __rbc_1_dummy -> true;

[__wake_rbc_1_connection_loss_1__] __pre_rbc_1_connection_loss_1 & __flag_rbc_1_connection_loss_1=0 ->
                                                    (__flag_rbc_1_connection_loss_1'=1);
[__sleep_rbc_1_connection_loss_1__] !__pre_rbc_1_connection_loss_1
                                & __flag_rbc_1_connection_loss_1=1 ->
                                                    (__flag_rbc_1_connection_loss_1'=0);
[__try_rbc_1_connection_loss_1__] __pre_rbc_1_connection_loss_1 & __flag_rbc_1_connection_loss_1=1 ->
                                        (__flag_rbc_1_connection_loss_1'=2) & (__rbc_1_z'=0);
[__dontfail_rbc_1_connection_loss_1__] __flag_rbc_1_connection_loss_1=2 ->
                                                    (__flag_rbc_1_connection_loss_1'=1);
[__fail_rbc_1_connection_loss_1__] __flag_rbc_1_connection_loss_1=2 ->
                    (0.1): (connected_1'=false)&(rbcclock_1'=4)&(__flag_rbc_1_connection_loss_1'=3)
                  + (0.2): (connected_1'=false)&(rbcclock_1'=3)&(__flag_rbc_1_connection_loss_1'=3)
                  + (0.4): (connected_1'=false)&(rbcclock_1'=2)&(__flag_rbc_1_connection_loss_1'=3)
                  + (0.2): (connected_1'=false)&(rbcclock_1'=1)&(__flag_rbc_1_connection_loss_1'=3)
                  + (0.1): (connected_1'=false)&(rbcclock_1'=0)&(__flag_rbc_1_connection_loss_1'=3);
[__restore_wake_rbc_1_connection_loss_1__] active_rbc_1_connection_loss_1
                                & __restorecond_rbc_1_connection_loss_1
                                & __pre_rbc_1_connection_loss_1 ->
                                                    (__flag_rbc_1_connection_loss_1'=1);
[__restore_sleep_rbc_1_connection_loss_1__] active_rbc_1_connection_loss_1
                                & __restorecond_rbc_1_connection_loss_1
                                & !__pre_rbc_1_connection_loss_1 ->
                                                    (__flag_rbc_1_connection_loss_1'=0);
```

```
    [] __rbc_1_reg_state & (!connected_1 & rbcclock_1>=MAX_TIME_NO_CONN) ->
                                                    (connected_1'=true) & (rbcclock_1'=0) & (__rbc_1_z'=0);
    [pos_1] __rbc_1_reg_state & !active_rbc_1_connection_loss_1 & (st_1=0 & rbcclock_1>=POS_INTERVAL) ->
                                                                (st_1'=1) & (rbcclock_1'=0) & (__rbc_1_z'=0);
    [MA_2] __rbc_1_reg_state & !active_rbc_1_connection_loss_1 & (st_1=1 & rbcclock_1>=MA_SENDING_TIME) ->
                                                                (st_1'=0) & (rbcclock_1'=0) & (__rbc_1_z'=0);
endmodule

module train_2

    braking_2: bool init false;
    toclock_2: clock;

    invariant
        (!braking_2=>toclock_2<=MA_RECV_TIMEOUT)
    endinvariant

    [pos_2] !braking_2 -> true;
    [BRAKE_2] !braking_2&toclock_2>=MA_RECV_TIMEOUT -> (braking_2'=true);
    [MA_2] true -> (toclock_2'=0);

endmodule
```

```
formula __pre_rbc_1_connection_loss_1 = (connected_1);
formula __restorecond_rbc_1_connection_loss_1 = (connected_1);
formula active_rbc_1_connection_loss_1 = (_flag_rbc_1_connection_loss_1=3);
formula __jump_rbc_1_connection_loss_1 = ((__pre_rbc_1_connection_loss_1
                                & __flag_rbc_1_connection_loss_1=0)
                               |(!__pre_rbc_1_connection_loss_1
                                & __flag_rbc_1_connection_loss_1=1)
                               |(__flag_rbc_1_connection_loss_1=2)
                               |(active_rbc_1_connection_loss_1
                                & __restorecond_rbc_1_connection_loss_1));

module rbc_1_connection_loss_1
   __s_rbc_1_connection_loss_1: [0..6] init 6;
   __x_rbc_1_connection_loss_1: clock;

   invariant
      (__s_rbc_1_connection_loss_1=1 | __s_rbc_1_connection_loss_1=3 | __s_rbc_1_connection_loss_1=4 =>
       __x_rbc_1_connection_loss_1<=1)
   endinvariant

   [__start__] __s_rbc_1_connection_loss_1=6 -> (__s_rbc_1_connection_loss_1'=0);
   [__wake_rbc_1_connection_loss_1__] __s_rbc_1_connection_loss_1=0 ->
                                 (__s_rbc_1_connection_loss_1'=1) & (__x_rbc_1_connection_loss_1'=0);
   [__sleep_rbc_1_connection_loss_1__] (__s_rbc_1_connection_loss_1=1|__s_rbc_1_connection_loss_1=4) ->
                                                                 (__s_rbc_1_connection_loss_1'=0);
   [__try_rbc_1_connection_loss_1__] __s_rbc_1_connection_loss_1=1 ->
```

```
                                                          (0.0005):(__s_rbc_1_connection_loss_1'=2)
                                                        + (1-(0.0005)):(__s_rbc_1_connection_loss_1'=3);
    [__fail_rbc_1_connection_loss_1__] __s_rbc_1_connection_loss_1=2 -> (__s_rbc_1_connection_loss_1'=5);
    [__dontfail_rbc_1_connection_loss_1__] __s_rbc_1_connection_loss_1=3 ->
                                                                (__s_rbc_1_connection_loss_1'=4);
    [] __s_rbc_1_connection_loss_1=4&__x_rbc_1_connection_loss_1=1 ->
                                    (__s_rbc_1_connection_loss_1'=1) & (__x_rbc_1_connection_loss_1'=0);
    [__restore_wake_rbc_1_connection_loss_1__] __s_rbc_1_connection_loss_1=5 ->
                                    (__s_rbc_1_connection_loss_1'=1) & (__x_rbc_1_connection_loss_1'=0);
    [__restore_sleep_rbc_1_connection_loss_1__] __s_rbc_1_connection_loss_1=5 ->
                                                                (__s_rbc_1_connection_loss_1'=0);

endmodule

formula __rbc_1_reg_state = !__rbc_1_dummy
                          & ((!__pre_rbc_1_connection_loss_1 & __flag_rbc_1_connection_loss_1=0)
                            |(__pre_rbc_1_connection_loss_1&__flag_rbc_1_connection_loss_1=1)
                            |(active_rbc_1_connection_loss_1&!__restorecond_rbc_1_connection_loss_1));
```

Referencias

[1] GNU General Public License. `http://www.gnu.org/licenses/gpl.html`.

[2] Wikipedia - European Train Control System. `http://en.wikipedia.org/wiki/European_Train_Control_System`.

[3] Miguel E. Andrés, Pedro R. D'Argenio, and Peter van Rossum. Significant diagnostic counterexamples in probabilistic model checking. In Hana Chockler and Alan J. Hu, editors, *Haifa Verification Conference*, volume 5394 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2008.

[4] Jean Arlat, Alain Costes, Yvez Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Tansactions on computers*, 42(8), August 1993.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A tutorial on Uppaal*.

[7] Eckard Böde, Mare Herbstritt, Holger Hermanns, Steve Johr, Thomas Peikenkamp, Reza Pulungan, Ralf Wimmer, and Bernd Becker. Compositional performability evaluation for STATEMATE. In *Third International Conference on the Qualitative Evaluation of Systems (QUEST'06)*. IEEE, 2006.

[8] Eckard Böde, Mare Herbstritt, Holger Hermanns, Steve Johr, Thomas Peikenkamp, Reza Pulungan, Ralf Wimmer, and Bernd Becker. Compositional performability evaluation for STATEMATE. *IEEE Transactions on Software Engineering*, 35(2), 2009.

[9] Alfredo Capozucca, Nicolas Guelfi, and Patrizio Pelliccione. The fault-tolerant insulin pump therapy. In *Workshop on Rigorous Engineering of Fault Tolerant Systems Event Information, in conjunction with Formal Methods 2005*, July 2005.

[10] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault injection: a method for validating computer-system dependability. *Computer*, 28(6), June 1995.

[11] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kennet L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6:217–232, 1995.

[12] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[13] Berteun Damman, Tingting Han, and Joost-Pieter Katoen. Regular expressions for pctl counterexamples. In *QEST*, pages 179–188. IEEE Computer Society, 2008.

[14] Richard de Neufville. The baggage system at Denver: prospects and lessons. *Journal of Air Transport Management*, 1(4):229–236, December 1994.

[15] Edsger W. Dijkstra. Notes on structured programming. Technical report, Technological University Eindhoven, The Netherlands, April 1970.

[16] Edgardo E. Hames. FALLUTO: Un model checker para la verificación de sistemas tolerantes a fallas. Master's thesis, Universidad Nacional de Córdoba, December 2009.

[17] Holger Hermanns, David N. Jansen, and Yaroslav S. Usenko. From StoCharts to MoDeST: a comparative reliability analysis of train radio communications. In *WOSP'05*, pages 13–23, 2005.

[18] Jin Jiang. Why does one need fault-tolerant control systems anyway? *Conference on Control and Fault Tolerant Systems*, October 2010.

[19] Israel Koren and C. Mani Krishna. *Fault tolerant systems*. Morgan Kauffman Publishers, 2007.

[20] Marta Kwiatkowska, Gethin Norman, and David A. Parker. Stochastic games for verification of probabilistic timed automata. In *Proc. 7th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'09)*, volume 5813 of Lecture Notes in Computer Science, pages 212–227. Springer, September 2009.

[21] Marta Kwiatkowska, Gethin Norman, David A. Parker, and Jeremy Sproston. Performance analysis of probabilistic timed automata using digital clocks. *Formal Methods in System Design*, 29:33–78, August 2006.

[22] Marta Kwiatkowska, Gethin Norman, Jeremy Sproston, and Fuzhi Wang. Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):1027–1077, July 2007.

[23] J. L. Lions. Ariane 5 - flight 501 failure. Technical report, the Inquiry Board, 1996.

[24] Kennet L. McMillan. *Symbolic Model Checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, May 1992.

[25] Ramiro Montealegre and Mark Keil. De-escalating information technology projects: Lessons from the Denver international airport. *MIS Quarterly*, 24(3):417–447, 2000.

[26] David A. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems.* PhD thesis, University of Birmingham, August 2002.

[27] Brian Randell. System structure for software fault tolerance. In *IEEE Transactions on software engineering*, volume SE-1, June 1975.

[28] Theo C. Ruys and Ed Brinksma. Model checking: Verification or debugging? Technical report, Faculty of Computer Science, University of Twente, 2000.

[29] J. Voas, G. McGraw, L. Kassab, and L. Voas. A "Cristal Ball" in software liability. *Computer*, 30(6), June 1997.

[30] Fuzhi Wang. *Symbolic Implementation of Model-Checking Probabilistic Timed Automata.* PhD thesis, University of Birmingham, September 2006.

[31] Fuzhi Wang and Marta Kwiatkowska. An MTBDD-based implementation of forward reachability for probabilistic timed automata. In *Proc. 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'05)*, volume 3707 of Lecture Notes in Computer Science, pages 385–399. Springer, October 2005.

[32] A. Zimmermann and G. Hommel. A train control system case study in model-based real time system design. In *International Parallel and Distributed Processing Symposium*, 2003.